Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Teste Baseado em Modelos Simbólicos para Sistemas de Tempo Real

Wilkerson de Lucena Andrade

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado
(Orientadora)

Campina Grande, Paraíba, Brasil

Resumo

Sistemas de tempo real são aqueles cujo correto comportamento não depende somente dos resultados gerados, mas também de quando os resultados são gerados. Sistemas de tempo real são utilizados em diferentes contextos, como por exemplo, monitoramento de pacientes em hospitais, controle de tráfego aéreo e sistemas embarcados em robôs, eletrodomésticos, veículos, etc. Para esses sistemas, confiança é uma importante propriedade que demanda uma aplicação rigorosa das atividades de V & V, pois defeitos podem significar perdas em termos financeiros, ambientais ou humanos. Como custos e consequências de falhas podem ser elevados, verificação formal e verificação de modelos têm sido utilizadas no processo de V & V. Entretanto, como essas abordagens possuem limitações práticas, teste também é utilizado como uma abordagem complementar porque permite a execução de cenários reais em ambientes de execução reais. Consequentemente, há um crescente interesse na busca por metodologias, técnicas e ferramentas para dar suporte ao teste de sistemas de tempo real, que por sua vez possui inúmeros desafios, tais como implementações compostas por atividades paralelas com eventos síncronos e assíncronos (interrupções), diferentes arquiteturas para instalação, limitação de recursos e restrições de tempo no ambiente de execução. Esta tese foca no teste de conformidade baseado em modelos para sistemas de tempo real. Nesse contexto, a maioria das abordagens atuais baseia-se em máquinas de estados ou em *timed automata*. Entretanto, a maioria dos sistemas de tempo real manipula dados enquanto estão sujeitos a restrições de tempo. A solução usual consiste em enumerar os valores de dados (em domínios finitos) enquanto o tempo é tratado de forma simbólica, levando ao problema da explosão do espaço de estados. Esta tese propõe um novo modelo para sistemas de tempo real que combina modelos simbólicos com *timed automata* a fim de tratar dados e requisitos de tempo de maneira simbólica. Uma teoria de teste de conformidade que lida com esse modelo é proposta juntamente com um processo de geração de casos de teste baseado na combinação de execução simbólica e *constraint solving* para tratar dados e análise simbólica para tratar aspectos temporais. Além disso, a abordagem proposta dá suporte ao teste de interrupções. Finalmente, dois estudos de caso são executados para avaliar a aplicação prática da abordagem proposta.

Abstract

Real-time systems are the ones whose correct behaviour depends not only on the generated results but also on whether the results are generated at the right time-points. Real-time systems are used in different contexts such as monitoring of patients in hospitals, air traffic control systems, and embedded systems in robots, appliances, vehicles, and so on. For these systems, dependability is an important property that demands rigorous application of V & V activities, since defects can mean losses in financial, environmental or human areas. As the costs and consequences of failures can be high, formal verification and model checking have been used in the V & V process. However, as these approaches have practical limitations, testing is also used as a complementary approach since it allows the execution of real scenarios within execution environments. Consequently, there is a growing interest in the search for methods, techniques and tools to support the testing of real-time systems, which poses a number of distinguishing challenges such as implementations composed of parallel activities with synchronous and asynchronous events (interruptions), with different deployment architectures, and resource limitation and time constraints on the execution environment. This thesis focuses on model-based conformance testing of real-time systems. In this context, current approaches are mostly based either on finite state machines/transition systems or on timed automata. However, most real-time systems manipulate data while being subject to time constraints. The usual solution consists in enumerating data values (in finite domains) while treating time symbolically, thus leading to the classical state explosion problem. This thesis proposes a new model of real-time systems as an extension of both symbolic transition systems and timed automata, in order to handle both data and time requirements symbolically. We propose a conformance testing theory to deal with this model and describe a test case generation process based on a combination of symbolic execution and constraint solving for the data part and symbolic analysis for timed aspects. Moreover, the proposed approach can deal with interruption testing. Finally, two case studies were performed in order to evaluate the practical application of the proposed approach.

Agradecimentos

Primeiramente a Deus por permitir-me concluir mais uma importante etapa da minha vida. Aos meus pais, Sebastião Solano de Andrade e Jucenilda de Lucena Andrade, que sempre me apoiaram de maneira incondicional e sempre estiveram presentes em todos os momentos em que precisei.

Um agradecimento muito especial a minha amada, Janaína, e a minha princesa, Gabriella, pela paciência, compreensão e pelos bons momentos juntos, incentivando-me muito durante o trabalho. Ao meu filho Luan, que mesmo antes de nascer também me deu muita força para chegar até aqui. As minhas irmãs, Wilkersya e Wilkerly, pela amizade e companheirismo.

Outro agradecimento especial a minha orientadora, Patrícia Duarte de Lima Machado, pela dedicada orientação, compreensão, apoio e incentivo. A Diego Almeida e a Augusto Macedo pelo apoio técnico, contribuindo diretamente para o sucesso do trabalho.

A equipe do projeto VerTeCs (INRIA Rennes - França) pelas discussões e importantes sugestões, especialmente a Thierry Jéron, Hervé Marchand e Nathalie Bertrand.

A equipe do projeto Motorola Brazil Test Center (Motorola BTCRD) por todas as discussões.

Aos professores Augusto Sampaio, David Déharbe, Tiago Massoni e Rohit Gheyi, pelos preciosos comentários visando melhorar a qualidade do trabalho. A CAPES, pelo apoio financeiro nos dois primeiros anos do doutorado. E por fim, a todas as pessoas que, direta ou indiretamente, contribuíram para o sucesso deste trabalho.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Os sistemas computacionais têm se tornado cada vez mais complexos e ubíquos, sejam através da Internet ou sistemas embarcados. Seguindo essa mesma linha de crescimento, um tipo especial de aplicações tem se tornado bastante comum, como é o caso dos sistemas militares, sistemas de controle de indústrias químicas e nucleares, sistemas de controle de tráfego aéreo, sistemas de monitoramento de pacientes em hospitais, sistemas embarcados em robôs, veículos, aviões, etc. Todas as aplicações citadas possuem uma importante característica em comum: requisitos temporais. O correto comportamento de tais aplicações não depende somente da integridade dos resultados obtidos (correção lógica ou *correctness*), mas depende também do tempo em que são produzidos (correção temporal ou *timeliness*) [79]. Sistemas computacionais com restrições de tempo são conhecidos como Sistemas de Tempo Real (STR).

Muitos STR são desenvolvidos com propósitos específicos e são fortemente acoplados ao hardware. Nesse caso, esses sistemas são conhecidos como Sistemas Embarcados de Tempo Real (SETR) [86]. Considerando os contextos de aplicações de STR citados anteriormente, podemos observar que muitos Sistemas de Tempo Real são complexos e críticos, visto que, problemas em STR podem significar perdas em termos financeiros, ambientais ou humanos. Para esses sistemas, confiança (*dependability*) é uma importante propriedade que demanda uma aplicação rigorosa das atividades de Verificação e Validação (V & V). Como custos e consequências de falhas podem ser elevados, verificação formal e verificação de modelos têm sido utilizadas no processo de V & V. Entretanto, essas abordagens possuem limitações práticas. Nesse contexto, teste surge como uma importante abordagem complementar pois a mesma permite a execução de cenários reais em ambientes de execução reais. Além disso, teste é uma das técnicas de validação mais populares atualmente e se utilizada de forma efetiva pode prover importantes indicadores de qualidade e confiabilidade de um produto. Assim, um dos desafios atuais está na busca por metodologias, técnicas e ferramentas para dar suporte ao teste de SERT.

Particularmente, o teste de SETR possui inúmeros desafios, como por exemplo: a plataforma de desenvolvimento é geralmente diferente da plataforma de execução, há várias plataformas de execução o que pode levar a existência de vários ambientes de desenvolvimento, geralmente os SERT são compostos por atividades paralelas com eventos síncronos e assíncronos (interrupções), há várias arquiteturas para instalação do sistema, há limitação de recursos e restrições de tempo no ambiente de execução. Nesse contexto, interrupções são usualmente aplicadas para a ativação de serviços imediatamente após as requisições. Para isso, a tarefa que está executando em primeiro plano é suspensa para liberar recursos para a tarefa que tratará a interrupção. Após o tratamento da interrupção, a tarefa interrompida continua sua execução a partir do mesmo ponto onde a mesma foi interrompida [2; 55; 79; 86]. Como exemplo da ocorrência de uma interrupção, considerando o contexto das aplicações para celulares, pode-se citar um cenário onde o usuário está compondo uma mensagem de texto e uma ligação chega a seu dispositivo, causando uma interrupção da aplicação de ligação na aplicação que envia mensagens de texto.

A maior parte dos trabalhos em V & V, no contexto de STR, está relacionada à verificação de modelos [13; 31; 54]. Verificação de modelos é uma técnica utilizada para verificar, de maneira automática e precisa, a corretude de modelos. Através da utilização de modelos e propriedades especificadas em vários formalismos, um verificador de modelos verifica se um modelo satisfaz uma determinada propriedade. Entretanto, se o mesmo rigor não for aplicado no teste da implementação do sistema, uma brecha é criada entre esses processos, o que permite a presença de defeitos na implementação mesmo que o modelo tenha sido verificado com sucesso.

Nesse sentido, várias abordagens foram desenvolvidas para adaptar técnicas de verificação de modelos para dar suporte à geração de casos de teste [19; 46; 60; 80]. Além disso, algumas abordagens clássicas de teste foram estendidas para dar suporte ao teste de STR, por exemplo, teste baseado em modelos [73], especialmente teste de conformidade [78] e o uso de propósitos de teste para selecionar cenários específicos a serem testados [17; 87]. Além disso, algumas abordagens estendem máquinas de estados finitos (MEF) e seus métodos associados para lidar com requisitos de tempo [46; 97].

Existem poucos trabalhos no contexto de teste de STR e a maioria deles utiliza (variações de) máquinas de estados ou (variações de) *timed automata* como modelo base. No entanto, a maioria das abordagens para teste de sistemas de tempo real somente abstrai tempo e enumeram valores de dados. Isso não é adequado quando uma especificação utiliza grandes ou infinitos domínios de dados, pois valores de dados são enumerados, levando ao problema da explosão do espaço de estados.

Na prática, os STR lidam com variáveis e ações com parâmetros. Assim, modelos mais poderosos são necessários, onde as variáveis, ações com parâmetros e tempo são explicitamente modelados e tratados de maneira simbólica. Há poucos trabalhos cujo objetivo é prover abordagens simbólicas para teste de software [27; 32; 52; 63; 64; 65; 72; 85; 108; 113; 114; 120]. Entretanto, a maioria dessas abordagens não considera requisitos de tempo [27; 32; 52; 63; 64; 65; 72; 85; 108] e os trabalhos mais recentes apresentados em [113; 114; 120] estão distantes no sentido de prover uma abordagem completa de teste. O trabalho descrito em [120] propõe um novo modelo simbólico juntamente com uma relação de conformidade simbólica, mas nem casos de teste são definidos formalmente nem algoritmos para geração de casos de teste são apresentados. A abordagem de teste de conformidade proposta em [113; 114] está restrita aos sistemas de tempo real baseados em fluxo de dados, dificultando o teste de interrupções. Além disso, casos de teste não são formalmente definidos, algoritmos não são apresentados e não há nenhuma ferramenta para dar suporte ao trabalho.

O restante deste capítulo está estruturado da seguinte forma: uma visão geral desta tese juntamente com suas principais contribuições é apresentada na Seção 1.1. A metodologia adotada está descrita na Seção 1.2. Finalmente, a Seção 1.3 apresenta a estrutura geral desta tese.

1.1 Visão Geral da Tese

O foco deste trabalho de doutorado está no teste de conformidade baseado em modelos simbólicos para sistemas de tempo real, onde a implementação é uma caixa preta cujos detalhes internos são desconhecidos. Assim, o testador só pode interagir com a implementação através do seu comportamento observável (entradas e saídas). Neste tipo de teste, casos de teste são derivados de uma especificação formal baseando-se em uma relação de conformidade entre a implementação e a especificação e são utilizados para guiar os veredictos da execução de testes [118]. Os veredictos são decididos por um componente chamado oráculo. Neste contexto, esta tese aborda as seguintes questões de pesquisa:

Questão de Pesquisa 1 *De que forma a teoria de teste baseado em modelos simbólicos pode ser estendida para poder testar sistemas de tempo real de maneira precisa?*

Questão de Pesquisa 2 *No contexto de teste baseado em modelos simbólicos para STR, como prover modelos capazes de especificar e testar eventos assíncronos, tais como interrupções, de maneira precisa?*

Questão de Pesquisa 3 *No contexto de teste baseado em modelos simbólicos para STR, é possível prover um oráculo automático?*

Para responder a estas questões de pesquisa, esta tese propõe uma nova abordagem de teste para sistemas de tempo real. Neste sentido, nossas principais contribuições são:

- Uma nova abordagem de teste de conformidade baseado em modelos para sistemas de tempo real é proposta, onde o sistema sendo testado é modelado através de uma combinação de modelos simbólicos com *timed automata*, lidando assim com a abstração de dados e requisitos de tempo;

- Um processo de geração de casos de teste que utiliza propósitos de teste como forma de seleção também é proposto. O processo é baseado em execução simbólica e *constraint solving* para tratar dados juntamente com análise simbólica para tratar aspectos temporais. Embora o modelo simbólico proposto possa representar ações internas e não-determinismo, os algoritmos definidos para a geração de casos de teste não consideram essas características. Além disso, quiescência e implementações não completas com relação ao conjunto de entradas estão fora do escopo desta tese;

- Uma estratégia para testar interrupções é proposta juntamente com uma maneira de definir propósitos de teste para checar cenários específicos com interrupções;

- A abordagem de teste proposta nesta tese fornece além de algoritmos para a geração de casos de teste, uma arquitetura de teste que inclui meios automáticos para a execução de testes e avaliação dos veredictos de forma confiável.

A aplicação prática da abordagem proposta é avaliada através de dois estudos de caso. Em um estudo de caso, todo o processo de teste é executado (a partir da geração de casos de teste até a execução dos mesmos), pois uma implementação do sistema está disponível. Por outro lado, como o segundo estudo de caso não possui uma implementação do sistema, apenas a geração de casos de teste é considerada. Os resultados obtidos mostram que a abordagem proposta reduz o esforço necessário para executar o processo de teste, considerando que a geração de casos de teste e avaliação dos resultados da execução são totalmente automatizadas. No entanto, alguns pontos de melhoria na atividade de execução de casos de teste foram detectados.

É importante ressaltar que esta tese considera as seguintes suposições, a fim de abordar as questões de pesquisa definidas:

1. A teoria de teste simbólico apresentada em [32; 63; 64; 65; 108] é sólida o suficiente para ser estendida para lidar com propriedades de tempo;

2. Teste baseado em modelos simbólicos provê uma boa base para o teste de conformidade em sistemas de tempo real.

1.2 Metodologia

A metodologia utilizada para desenvolver este trabalho está descrita a seguir:

- O primeiro passo foi realizar uma revisão dos trabalhos voltados para teste baseado em modelos para sistemas de tempo real com o objetivo de identificar os problemas em aberto. É importante mencionar que esta revisão foi constantemente atualizada durante o desenvolvimento deste trabalho;

- O problema de testar de interrupções foi investigado em um contexto sem considerar tempo. Neste caso, uma abordagem de teste interrupção foi proposta para sistemas reativos;

- Execução de um estudo de caso para avaliar a abordagem de teste de interrupção proposta;
- Abordagens baseadas em modelos simbólicos foram estudadas e uma abordagem foi escolhida como teoria base;
- Uma estratégia de teste de interrupção foi definida para a abordagem baseada em modelos simbólicos escolhida como teoria base;
- O formalismo baseado em modelos simbólicos escolhido foi estendido para lidar com requisitos de tempo e sua semântica foi formalmente definida;
- Formalização dos conceitos de caso de teste e propósito de teste;
- Uma relação de conformidade existente foi escolhida para ser utilizada na abordagem proposta;
- Definição de algoritmos para a geração e seleção de casos de teste;
- Formalização da noção de veredictos, considerando que a execução de um caso de teste pode produzir um dos seguintes resultados: aprovado, falha ou inconclusivo;
- Definição de estratégias para a especificação e teste de interrupções utilizando a abordagem proposta;
- Definição de uma arquitetura de teste e hipóteses de controlabilidade para a execução de casos de teste;
- Desenvolvimento de um ambiente para permitir a execução de testes e avaliação dos resultados de maneira automática;
- Execução de estudos de caso para avaliar a aplicabilidade da abordagem proposta.

1.3 Estrutura da Tese

As demais partes deste documento estão estruturadas da seguinte forma:

Capítulo 2: Fundamentação Teórica Este capítulo fornece o embasamento teórico necessário para entender este trabalho, incluindo conceitos da área de Teste de Software tais como casos de teste, oráculos e técnicas de teste. Finalmente, conceitos inerentes aos sistemas de tempo real são apresentados.

Capítulo 3: Teste de Interrupção em Sistemas Reativos Este capítulo apresenta uma abordagem de teste de conformidade para sistemas reativos com interrupções. Sistemas de tempo real não são considerados neste capítulo.

Capítulo 4: Trabalhos Relacionados e Problemas Identificados Este capítulo apresenta uma revisão dos trabalhos relevantes no contexto de teste de sistemas de tempo real. No final, alguns problemas em aberto são descritos.

Capítulo 5: Timed Input-Output Symbolic Transition Systems Este capítulo propõe o formalismo simbólico definido com o objetivo de abstrair dados e tempo na especificação de sistemas de tempo real.

Capítulo 6: Teste de Conformidade com TIOSTS Este capítulo apresenta a teoria de teste de conformidade considerando com o modelo proposto. Além disso, o processo de geração de casos de teste é descrito juntamente com uma discussão acerca de algumas propriedades dos casos de teste gerados.

Capítulo 7: Teste de Interrupção em Sistemas de Tempo Real A estratégia de modelagem e teste de interrupções utilizando o formalismo simbólico proposto é descrita neste capítulo.

Capítulo 8: Estudos de Caso Este capítulo apresenta uma demonstração prática da abordagem baseada em modelos simbólicos proposta para sistemas de tempo real.

Capítulo 9: Considerações Finais Este capítulo final apresenta as considerações finais e as perspectivas para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como objetivo principal fornecer o embasamento teórico necessário para entender os conceitos empregados nesta tese. Serão apresentados os principais conceitos da área de Teste de Software, destacando o teste baseado em modelos e teste simbólico, e também os conceitos inerentes aos sistemas de tempo real.

2.1 Software Testing

Software testing is an activity that involves the effort to find evidences of defects inserted into the software during any phase of development or maintenance of software systems. These defects may be due to omissions, inconsistencies or misunderstanding of requirements or specifications by developers [95]. In the software testing context, some concepts are widely used: **failure**, **fault**, and **error**. According to Binder [18], a **failure** is the manifested inability of a system to correctly perform a required function; a **fault** is defined as the absence of code or the presence of incorrect code in a computer program that causes the failure; and **error** is a human action that results in a software fault.

Testing is an important activity that contributes to ensuring that a software system does everything it is supposed to do. Some testing efforts extend the focus to ensure an application does nothing more than it is supposed to do. In any case, testing provides means to assess the existence of defects (faults) which could result in a loss of time, property, customers, or life [95].

For a long time, the software testing process was defined within software development processes as a disconnected activity that was only taken into account at the end of the development processes. This traditional view is considered as being inefficient because of high costs associated with correction of detected errors and maintenance of software. This has contributed to the development of methods and systematic testing techniques where the testing activities are applied in parallel during the development process [95].

There are several kinds of tests that can be applied depending on the property of the systems to be tested (for instance, interface, performance, safety, etc.), and their type (for instance, object-oriented software, distributed systems, reactive systems, real-time systems).

The remainder of this section presents several concepts related to software testing and that are important to understand this thesis, such as test cases, oracles, approaches to identify and generate test cases, conformance testing, and some testing techniques such as model-based testing, property oriented testing, and symbolic testing.

2.1.1 Test Cases

A test case is a set of inputs, execution conditions, and expected results chosen in order to test a particular behaviour of a system [18]. The main key of software testing is to determine a set of test cases (named test suite) for the software system to be tested. Every test case must have at least the following information [67]:

- Inputs
  - Conditions that must be satisfied before the test execution;
  - The actual inputs chosen to test the system;
- Outputs
  - Postconditions that must be satisfied after the test execution;
  - The actual output produced by the system under test.

Moreover, a good test case must present additional information for supporting the testing management [67]. For example, a test case may have a unique identifier, a purpose, an execution history, etc. Considering all this information, the act of testing consists in satisfying the preconditions, providing the test case inputs, observing the outputs, and then comparing these outputs with the expected outputs to decide whether the test pass or not.

In the context of this work, test cases can be classified into two types: **instantiated** and **abstract** test cases. We say that a test case is **instantiated** when the values of all variables needed for the test execution are properly assigned during the test case generation process, whereas we say that a test case is **abstract** when it has variables with unassigned values. In the latter case, the tester must assign values, according to the preconditions, during the test execution.

2.1.2 Oracles

The execution of a test case emits a pass verdict when the system produces an acceptable result. In order to decide which verdict must be emitted, an evaluation is made by comparing the actual result with an expected result. The component responsible for performing this evaluation is called test oracle or simply oracle [18]. Thus, an oracle is a mechanism that applies a pass or fail verdict to a system execution [105]. For this, it is necessary a result generator and a comparator. The former is responsible for generating the expected results for an input and the latter has the objective of checking the actual results against the expected results.

Considering that a test oracle is a generation and comparison mechanism, it can be classified into three types: manual, automated, and partially automated [18]. Considering the manual oracle, both generation and comparison are manually performed. In the automated oracle, both generation and comparison are automatically performed. Finally, in the partially automated oracle, one of the activities is manually performed, whereas the other is automatic.

Several artefacts developed during the development process can support the testing process as an oracle. The system specification can be used as an oracle, a table of examples of inputs and expected outputs or simply the knowledge of how the software system should operate provided by the development team can be also used as an oracle [18].

In practice, manual and partially automated oracles are error-prone. If a system under test fails to provide some functionality in a very common situation (for example, a menu option is in a wrong place, the system is aborted with an exception or it is restarted), then maybe it can be seen to have a fault. But considering an expected output, specified only by an imprecise description in natural language, a tester may fail to notice a failure. To do better, an oracle must be automated. But, due to the semantic gap between specification and real application values, a problem named the oracle problem arises when an automated mechanism to emit verdicts cannot be defined [91; 92].

The concepts of test case, test suite, and oracle can be related based on a formal framework [56; 91]. Let IUT be an implementation of a software system under test whose input domain is D and output domain is X. Let TC be a test case which is defined as a total function from elements of $D' \subseteq D$ to elements of $X' \subseteq X$. Then $\mathit{dom}(TC) = D'$ denotes the domain of TC. Also, let $TC(p)$ be the corresponding expected output for a given input $p \in \mathit{dom}(TC)$ and $\text{IUT}(p)$ denote the actual result of executing IUT with input p. The TC passes the system IUT if and only if it passes IUT on all inputs in $\mathit{dom}(TC)$, that is, $\text{IUT}(p) = TC(p)$ for all $p \in \mathit{dom}(TC)$. As $\mathit{dom}(TC)$ is likely to be infinite, a finite test suite $T \subseteq \mathit{dom}(TC)$ needs to be selected. A function O is called an oracle for IUT on TC if for all $p \in D$ [91]:

$$O(p) = \begin{cases} true, & \text{if } \text{IUT}(p) = TC(p) \\ false, & \text{if } \text{IUT}(p) \neq TC(p) \\ true, & \text{if } p \notin \mathit{dom}(TC) \end{cases}$$

Considering the oracle problem, it arises because of the limitation in the definition of an effective procedure to compare $\text{IUT}(p)$ with $TC(p)$, mainly because these values are defined at different levels of abstraction [91].

2.1.3 Test Cases Identification

The two fundamental approaches used to identify test cases are known as structural and functional testing [67]. Each of these approaches has its advantages and disadvantages.

Structural testing, also known as white box testing, is a kind of testing where test cases are identified based on the system implementation. The objective of structural testing is to test procedural details [101]. Because structural testing is based on the implementation, it can test parts of the system that are not in the specification, but, on the other hand, the structural testing fails to identify behaviours which are in the specification, but have not been implemented.

Functional testing (also known as black box testing) is a kind of testing based on the view that the software system can be considered as a function that maps values from its input domain to values in its output domain [67], that is, a kind of testing performed to verify whether, for a given input, the system produces the correct output. Functional testing is performed only based on the specification of the system.

Because functional test cases are only identified based on the specification, they are independent of how the system is implemented, unlike structural test cases, and therefore, even if the implementation of the system is changed, the test cases are still useful. Another important advantage is that testing activities can be performed in parallel with the implementation, contributing to a better understanding and correction of models and specifications from initial stages of the development processes, avoiding late detection of problems, thus reducing the impact and costs associated with the changes.

In addition to the classification of tests following the fundamental approaches, structural and functional, we can make a new distinction with respect to several aspects of the behaviour of the system to be tested. When the specification is defined by models, the approach is called model-based testing [43][1]. When the test is carried out to verify whether the system has the planned functionalities and if those functionalities are in accordance with the specification, it is called conformance testing [117]. When the goal is to test specific properties of the system, test case generation can be guided by informal descriptions of the behaviours to be tested. In this case, the approach is called property oriented testing [62].

2.1.4 Model-Based Testing

In the last decade, perhaps due to the popularization of object-oriented programming and use of models in software engineering, there was a great development of a testing technique known as model-based testing (MBT). MBT is a general term used to name a set of techniques based on models of applications being tested in order to perform activities of test [43]. Such activities can be either generation of test cases or evaluation of test results.

The main activities related to model-based testing, shown in Figure 2.1, are described below:

Build the model: the model is built from the requirements of the software system under

[1] As models are considered as specifications in this thesis, these terms are used interchangeably.

test.

Generate test cases: test cases are extracted from the model with the objective of evaluating whether the system is in accordance with its requirements.

Generate test oracle: the test oracle is generated based on the model. The test oracle is responsible for deciding which outputs indicate the correct behaviour of the system, that is, the expected results.

Run tests: the application is exercised with generated test cases, producing new outputs.

Compare actual outputs with expected outputs: the outputs of the system under test, obtained in the previous step, are compared by the test oracle with the expected outputs.

The process of model-based testing begins when the requirements of the software system are defined. From requirements, a model that represents the expected behaviour of the system is built. After defining the model, the next step is the generation of test cases. The specification of test cases includes, among other information, expected inputs and outputs. Using these inputs, the system is executed and its behaviour is observed. The last step is to compare the obtained outputs with the expected outputs to assess whether or not the system is in accordance with requirements.

One of the main advantages of using MBT is that the generated model can serve as a reference point for communication between all the people involved in the development process. Another important advantage is that the most popular models have a rich theoretical basis that facilitates the generation and automation of the testing process [43].

One drawback of using MBT is the need of knowing the notation of the model and the theoretical basis to take the most of the model chosen. To make the team acquire the necessary knowledge implies investment in training and lack of time, in addition to time spent on the construction of the model [43]. Another disadvantage is the high dependence with respect to the model, that is, as the test activities are carried out based on the model of the system, the quality of testing is directly related to the quality of the model.

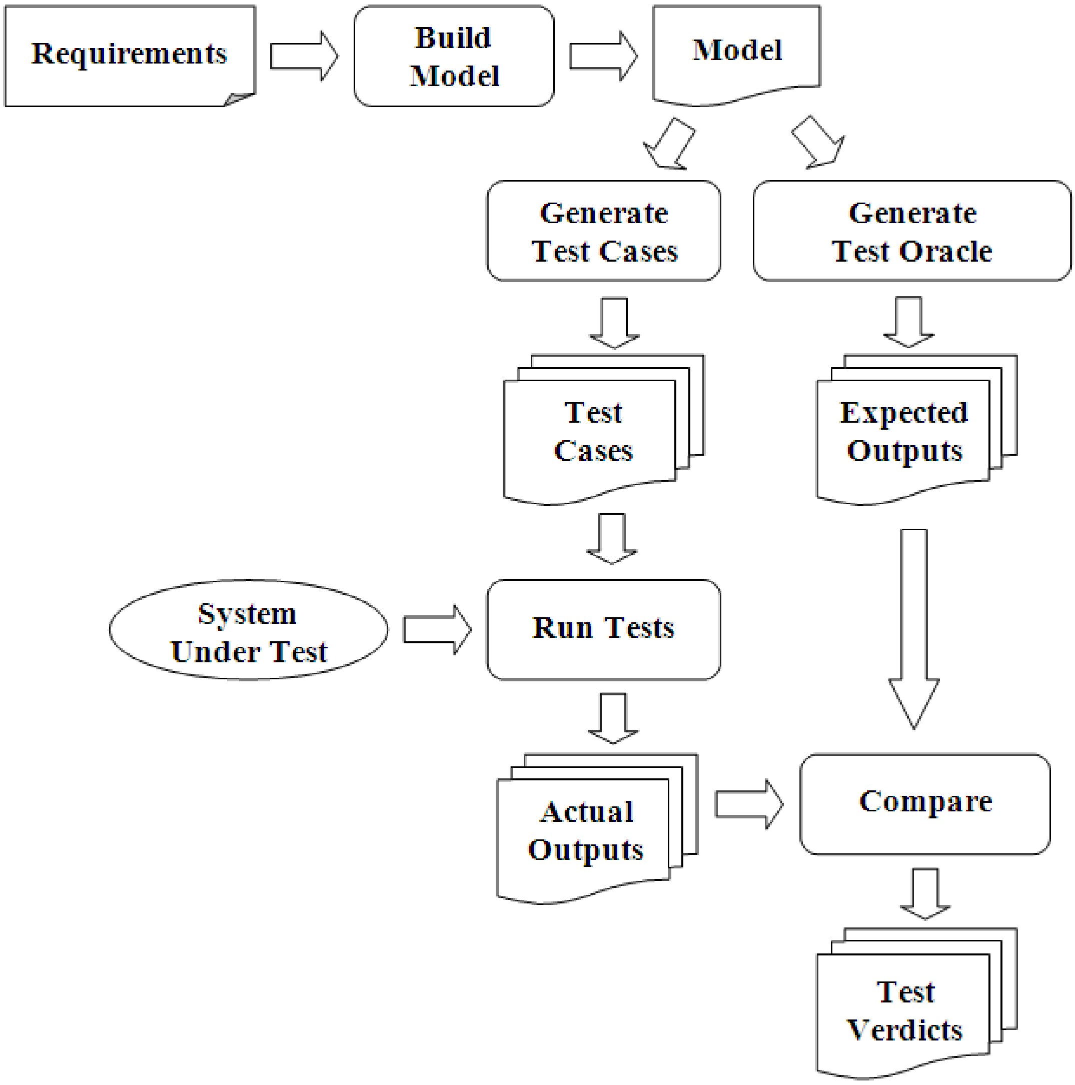


Figura 2.1: MBT Activities

2.1.5 Conformance Testing

Conformance testing is a kind of testing used to verify whether the implementation of a software system is in accordance with the specification of its functional behaviour. So, this subsection presents, in a formal manner, a conformance testing approach based on the framework proposed by Tretmans [118]. Therefore, it is important to link the informal world of implementations and tests with the formal world of specifications and models.

Conformance testing relates a specification and an implementation under test (IUT) by the relation **conforms-to** $\subseteq IMPS \times SPECS$, where $IMPS$ represents the universe of implementations and $SPECS$ represents specifications. Then, IUT **conforms-to** s if and only if IUT is a correct implementation of s.

The **conforms-to** relation is hard to be checked by testing and the implementations are generally unsuitable for formal reasoning. Therefore, a test hypothesis is assumed where any IUT can be modelled by a formal object $i_{\text{IUT}} \in MODS$, where $MODS$ represents the universe of models [15]. Then, an implementation relation **imp** $\subseteq MODS \times SPECS$ is defined such that IUT **conforms-to** s if and only if i_{IUT} **imp** s.

Let $TESTS$ be the domain of test cases and $t \in TESTS$ be a test case. Then $\text{EXEC}(t,\text{IUT})$ denotes the operational procedure of applying t to IUT. This procedure represents the test execution. Let an observation function that formally models $\text{EXEC}(t,\text{IUT})$ be defined as $obs : TESTS \times MODS \rightarrow \mathcal{P}(OBS)$. Then, $\forall$ IUT $\in IMPS\ \exists i_{\text{IUT}} \in MODS\ \ \forall t \in TESTS\cdot$ $\text{EXEC}(t,\text{IUT}) = obs(t, i_{\text{IUT}})$, according to the test hypothesis.

Let a family of verdict functions $v_t : \mathcal{P}(OBS) \rightarrow \{$**fail**, **pass**$\}$ which can be abbreviated to IUT **passes** $t \Leftrightarrow_{def} v_t(\text{EXEC}(t,\text{IUT})) =$ **pass**. Then, for any test suite $T \subseteq TESTS$, IUT **passes** $T \Leftrightarrow \forall t \in T\cdot$ IUT **passes** t. Also, IUT **fails** $T \Leftrightarrow \neg($IUT **passes** $T)$. A test suite that can distinguish between all conforming and non-conforming implementations is called *complete*. Let $T_s \subseteq TESTS$ be complete. Then, IUT **conforms-to** s if and only if IUT **passes** T_s.

A complete test suite is a very strong requirement for practical testing. Then, weaker requirements are needed. A test suite is *sound* when all correct implementations and possibly some incorrect implementations pass it, that is, any detected faulty implementation is non-conforming, but not the other way around. Let $T \subseteq TESTS$ be sound. Then, IUT **conforms-to** $s \Rightarrow$ IUT **passes** T. The other direction of the implication is called *exhaustiveness*, meaning that all non-conforming implementations will be detected.

In practice, sound test suites are more commonly accepted, since rejection of conforming implementations, by exhaustive test suites, may lead to unnecessary debugging. Let $der_{imp} : SPECS \rightarrow \mathcal{P}(TESTS)$ be a test suite derivation algorithm. Then, $der_{imp}(s)$ should only produce sound and/or complete test suites.

This testing framework is instantiated by several works using the most different notations. For instance, Tretmans [118] instantiated the framework with Labelled Transition Systems (LTS), Jard and Jéron [62] instantiated it using Input-Output Labelled Transition Systems (IOLTS), Larsen et al. [80] and Krichen [73] instantiated the framework using Timed Input-Output Transition Systems (TIOTS), Briones and Brinksma [23] extended the Tretmans' framework for real-time systems, etc.

One of the most important properties considered in conformance testing is called quiescence. In practice, tests observe the behaviour of the system and its quiescence, that is, the absence of outputs. Quiescence is observed using timers, for instance, whenever a tester sends an input to the implementation, a timer is reset. The duration of the timer is chosen such that, if no output occurs while the timer is running, then no output will ever occur. Then, when the timer finishes, the tester can conclude that the implementation is quiescent. This approach avoids the rejection of implementations expected to be quiescent in some points and rejects the implementations that are not, ensuring the soundness of the test cases.

In order to distinguish between observations of quiescence that are allowed by a specification and those that are not, all possible points where an implementation may become quiescent must be made explicit in the specification. There are three possibilities of quiescence: deadlock, output quiescent, and livelock [62]. A deadlock state is a state where the system cannot evolve anymore. An output quiescent state is a state where the system is waiting only for an input from the environment. And, livelock is related to the loops of internal actions, that is, loops of actions that are not seen from the environment.

2.1.6 Property Oriented Testing

It is important to note the difference between testing for conformance and testing from test purposes. The former aims to accept/reject a given implementation. On the other hand, the latter aims to observe a desired behaviour that is not necessarily directly related to a required behaviour or correctness. If this desired behaviour is observed then confidence on correctness may increase. Otherwise, no definite conclusion can be based solely on this information. Due to its overloaded use, test purpose is called observation objective in [41]. Nevertheless, the term test purpose is kept in thesis. The concepts introduced in Subsection 2.1.5 are extended for test purposes in this subsection.

Test purposes describe desired observations that we wish to see from the implementation during the test. Test purposes are related to implementations that are able to exhibit them by a well chosen set of experiments. This is defined by the relation **exhibits** $\subseteq IMPS \times TOBS$, where $TOBS$ is the universe of test purposes. To reason about exhibition, we also need to consider the test hypothesis from Subsection 2.1.5 by defining the *reveal* relation **rev**

$\subseteq MODS \times TOBS$, so that, for $e \in TOBS$, IUT **exhibits** e if and only if i_{IUT} **rev** e, with $i_{\text{IUT}} \in MODS$ of IUT.

Let a verdict function $H_e : \mathcal{P}(OBS) \to \{$**hit**, **miss**$\}$ which can decide whether a test purpose is exhibited by an implementation. Then, IUT **hits** e **by** $t_e =_{def} H_e(\text{EXEC}(t_e,\text{IUT})) =$ **hit**. This is extended to a test suite T_e as IUT **hits** e **by** $T_e =_{def} H_e(\bigcup\{\text{EXEC}(t,\text{IUT}) \mid t \in T_e\}) =$ **hit**, which differs from the **passes** abbreviation.

An *e-complete* test suite can distinguish among all exhibiting and non-exhibiting implementations, such that, IUT **exhibits** e if and only if IUT **hits** e **by** T_e. An *e-exhaustive* test suite can only detect non-exhibiting implementations (IUT **exhibits** e implies IUT **hits** e **by** T_e), whereas an *e-sound* test suite can only detect exhibiting implementations (IUT **exhibits** e if IUT **hits** e **by** T_e). Note that the purpose of the *sound* test suites and *e-sound* test suites are similar, even though the implications are relatively inverted. *Sound* test suites can reveal the presence of faults, whereas the *e-sound* can reveal intended behaviour.

Conformance and exhibition can be related aiming to consider test purposes in test selection to obtain test suites that are sound and e-complete. We want *e-soundness* so that we can conclude that a hit result always implies exhibition, whereas we require *e-exhaustiveness* because we want to be able to find all implementations that are able to exhibit. Soundness provides us with the ability to detect non-conforming implementations. Contrary to complete test suites, *e-complete* test suites are feasible.

Finally, it is important to remark that both conforming and non-conforming implementations may reveal a test purpose. An ideal situation, where all correct implementations also exhibit, would be to only consider a test purpose e when i **rev** $e \supseteq i$ **passes** T, where $T \subseteq TESTS$. However, this situation is not practical. Test purposes are chosen so that: $\{i \mid i$ **rev** $e\} \cap \{i \mid i$ **imp** $s\} \neq \emptyset$. In this case, a test execution with test case $T_{s,e}$ that is both sound and e-complete and that results in **fail** means non-conformity, since sound test cases do not reject conforming implementations and e-complete test cases distinguish between all exhibiting and non-exhibiting implementations. Also, if the result is $\{$**pass**, **hit**$\}$, confidence on correctness is increased, as the hit provides possible evidence of conformance.

2.1.7 Test Case Generation

The test case generation activity can be done following two different approaches: offline or online testing. In offline testing, all test cases are extracted from the specification, after that, they are executed against the implementation to obtain a verdict.

The other approach to test case generation is online (on-the-fly) testing where the generation and execution are combined into a single step. In this approach, a single action or input is extracted from the specification at a time and it is immediately executed on the implementation. Then the output produced by the implementation is checked against the specification. After that, another action or input is extracted again and so forth until the end of the test, or until a defect (fault) is detected.

As advantages, the offline test generation can be guided to generate test cases in order to reach some objective, e.g. specification structural coverage, test specific behaviours with test purposes, and so on. During the offline test generation, all time constraints are resolved before the execution, so the generated test cases are cheaper and faster to execute [59].

One of the main advantages of online testing is that the classical state space explosion problem is reduced because only a limited part of the state space needs to be stored at any point in time, whereas the state space explosion problem is very common when the offline approach is adopted because the state space needs to be entirely built and stored. Another advantage of online testing is that some important characteristics of real implementations as non-determinism can be treated during the test execution more easily. However, online testing may not be applicable in environments with limited resources, for example in the test of some embedded systems (e.g. smart cards, mobile phones, music players, etc), because of the need of very efficient test generation algorithms and many resources to execute them.

2.1.8 Symbolic Execution

Symbolic execution is a technique for analysing programs which represents program inputs with symbolic values instead of concrete values [33; 71]. The execution of programs is simulated by manipulating expressions involving symbolic values. Thus, the outputs are expressed as a function of the symbolic inputs. This technique is used in different contexts such as test input generation, reachability analysis, partial correctness proving of programs, Algorithm 2.1: Code for Swapping Two Integers Variables

```
1 int x, y;
2 if (x > y) {
3   x = x + y;
4   y = x - y;
5   x = x - y;
6   if (x - y > 0) {
7     assert(false);
8   }
9 }
```

etc.

The execution paths of a program identified during its symbolic execution are represented as a symbolic execution tree, whose nodes are the program states connected by program transitions. A program state includes the symbolic values of program variables, a path condition (PC), and a program counter. A path condition is a (quantifier-free) boolean formula over the symbolic inputs. When a PC is satisfiable means that it is possible to reach the specific program point associated with it. Otherwise, the referred specific program point is unreachable.

Consider the code fragment in Algorithm 2.1, which swaps the values of integer variables x and y when x is greater than y, with its corresponding symbolic execution tree presented in Figure 2.2, where transitions are labelled with program statement line numbers [104]. At the beginning, *PC* is true and x and y have X and Y as symbolic values, respectively. At each statement of the code, *PC* is updated according to conditions associated with variables. Analysing the symbolic execution tree of Figure 2.2, it is possible to conclude that line 7 of Algorithm 2.1 is unreachable because the corresponding *PC* is not satisfiable. In practice, constraint solvers are used to verify whether a path condition is satisfiable.

2.1.9 Symbolic Testing

In the last years, several theories and techniques of test case generation have been developed through specifications modelled by variations of the classic LTS [12; 29; 40; 62; 85; 117]. Basically, LTS models and its variations represent a system behaviour through a graph where

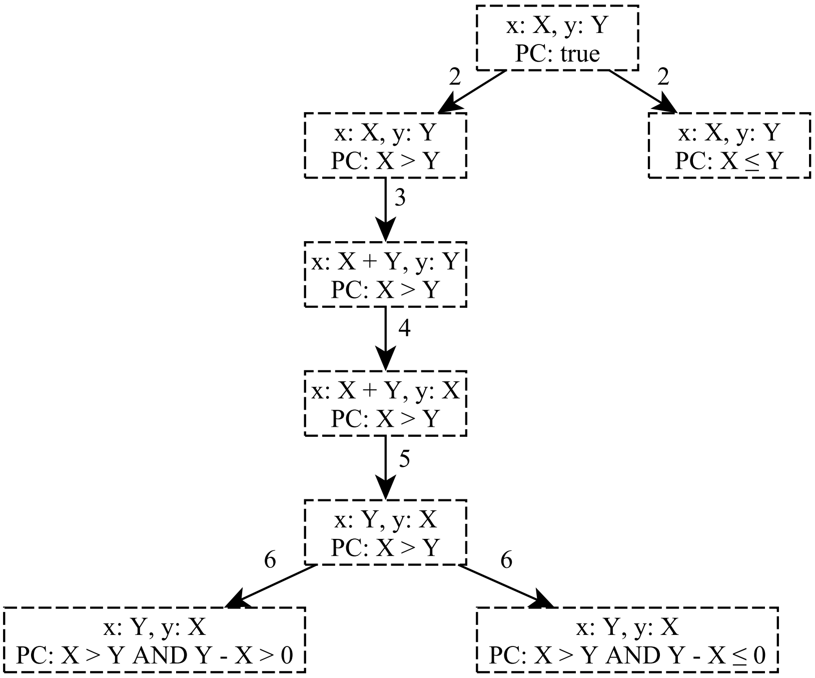


Figura 2.2: Symbolic Execution Tree of Algorithm 2.1

the states are the possible system configurations and the edges represent the action of moving between these configurations through actions occurrence.

However, LTS models are not suitable when the specification uses large or infinite data domains because each value in the data domain is represented as a system state, leading to the classical state space explosion problem. Consequently, many tools can only be used in very restricted and finite domains.

In practice, test cases (written, for example, using TTCN [49]) can be real programs with parameters and variables. In this context, a new approach to testing arises: symbolic testing. Symbolic testing is a testing approach based on powerful models where variables and parameters are explicitly modelled and treated in a symbolic way.

In the context of symbolic models, the principle is to adapt some existing approach to, beyond representing the system behaviour, represent the system data without enumerating the data values. There are still very few works in this context and most of them use (variations of) state machines or labelled transition systems as the underlying model.

State machines tend to be used in synchronous contexts, where inputs and outputs appear together in a single transition. Thus, they are unsuitable for representing some characteristics of reactive systems such as non-determinism and interruptions. Considering the use of state machines, there is a tool named GAST [72] that was extended to deal with Extended Finite State Machines (EFSM) specifications. In this tool, properties and data types are expressed in first-order logic, and based on this information, test data is automatically generated. This tool is less suitable for testing because the concept of state is not present in a clear manner, even though it is possible to represent states defining explicitly a complex data structure that represents the state space. GAST's algorithm unfolds, in an on-the-fly way, the data type structure in order to select a path in the EFSM.

The use of (variations of) labelled transition systems is more common in the literature. Lestiennes and Gaudel [85] developed a strategy of test generation and selection based on selection hypotheses combined with an operation of unfolding algebraic data types and predicate resolution. Frantzen et al. [52] extended the theory presented in [117] to support software testing based on symbolic models. The symbolic framework developed by Frantzen et al. uses concepts from first-order logic as underlying theory for dealing with guards and variables, quiescence is taken into account, and some ideas about coverage is discussed. However, the symbolic framework does not consider test purposes and there is no tool support. Calamé et al. [27] proposes an approach combining symbolic models, data abstraction, and constraint solving to generate test cases. The main idea is to apply data abstraction to abstract the model in a finite state one, use the TGV tool [62] to generate abstract test cases, and finally, constraint solving is applied to instantiate the test cases.

One of the most solid approaches in the context of symbolic testing is presented in [32; 63; 64; 65; 108]. This method works directly on high-level specifications given as Input-Output Symbolic Transition Systems (IOSTS) without enumerating their state space. Test purposes are taken into account to verify specific behaviours of an implementation. Approximate coreachability analysis is used to prune paths potentially not leading to pass verdicts [63]. The coreachability analysis is based on Abstract Interpretation [36] and the concept of test generation with verification techniques is also based on the theory presented in [62; 117]. Finally, constraint solving is applied to instantiate the test cases. Moreover, all the symbolic testing approach is supported by the STG tool [32]. So, the theory related to IOSTS will be presented in more detail below.

The IOSTS is a model of extended labelled transition systems that was inspired by I/O automata [89]. An IOSTS is a symbolic automata with a finite set of locations, typed variables, and the communication with its environment is performed through actions carrying parameters.

Definition 2.1 (IOSTS). *Formally, an IOSTS is a tuple $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$, where [108]:*

- *V is a finite set of typed variables;*
- *P is a finite set of parameters. For $x \in V \cup P$, $type(x)$ denotes the type of x;*
- *Θ is the initial condition, a predicate with variables in $V \cup P$;*
- *L is a finite, non-empty set of locations;*
- *$l^0 \in L$ is the initial location;*
- *$\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is a finite, non-empty alphabet, where $\Sigma^?$ is a finite set of input actions, $\Sigma^!$ is a finite set of output actions, and Σ^τ is a finite set of internal actions. Each action $a \in \Sigma$ has a signature $sig(a) = \langle p_1, ..., p_k \rangle$, that is a tuple of distinct parameters. The signature of internal actions is the empty tuple;*
- *$\mathcal{T}$ is a set of transitions, where each transition consists of:*
  - *a location $l \in L$, called the origin of the transition,*
  - *an action $a \in \Sigma$, called the action of the transition,*
  - *a predicate G with variables in $V \cup P \cup sig(a)$, called the guard,*
  - *an assignment A, such that for each variable $x \in V$ there is exactly one assignment in A, of the form $x := A^x$, where A^x is an expression on $V \cup P \cup sig(a)$,*
  - *a location $l' \in L$, called the destination of the transition.*

◇

Figure 2.3 shows an example of an IOSTS. In graphical representations, input actions are followed by the "?" symbol and output actions are followed by the "!" symbol. These symbols are used only as notation, they are not part of the action's name. The simple IOSTS depicted in Figure 2.3 models the triangle problem, an example widely used in the literature [67; 101; 106]. The input of the problem is three integers representing the sides of a triangle and the output is the type of the triangle. At the beginning, the system is in the

Idle location. Next, the system expects the *Read* input carrying three strictly positive integer parameters *p*, *q*, and *r*. Then the values of the parameters are saved into the variables *a*, *b*, and *c*, respectively. If the values of the variables do not represent a triangle the system leaves the *CheckTriangle* location and goes to the *End* location emitting the *NotATriangle* output. Otherwise, the system emits the *IsTriangle* output followed by the type of the triangle (equilateral, isosceles, or scalene).

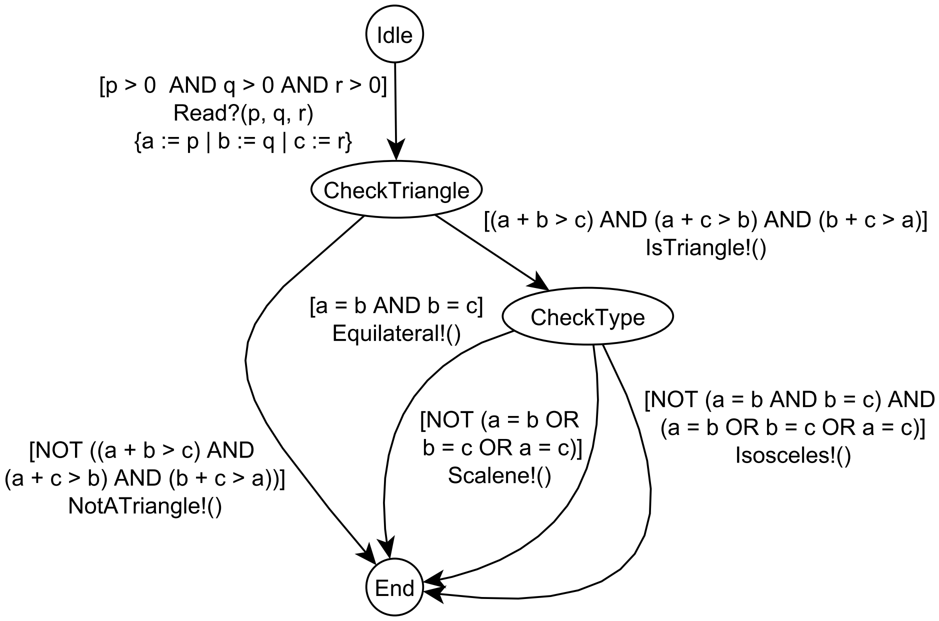

Idle
[p > 0 AND q > 0 AND r > 0]
Read?(p, q, r)
{a := p | b := q | c := r}
CheckTriangle
[(a + b > c) AND (a + c > b) AND (b + c > a)]
IsTriangle!()
CheckType
[a = b AND b = c]
Equilateral!()
[NOT (a = b AND b = c) AND
(a = b OR b = c OR a = c)]
Isosceles!()
[NOT (a = b OR
b = c OR a = c)]
Scalene!()
[NOT ((a + b > c) AND
(a + c > b) AND (b + c > a))]
NotATriangle!()
End

Figura 2.3: IOSTS Example

The semantics of IOSTS is defined through *Input-Output Labelled Transition Systems* (IOLTS) [35]. An IOLTS is a variant of the classic LTS that makes distinction between events of the system that are controllable by the environment (the inputs) and those that are only observable (the outputs) [117]. Moreover, internal actions can be represented too.

Definition 2.2 (IOLTS). *An IOLTS is a tuple $\langle Q, Q^0, \Lambda, \rightarrow \rangle$, where [35]:*

- *Q is a possibly infinite set of states;*
- *$Q^0 \subseteq Q$ is the possibly infinite set of initial states;*
- *$R = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ is a possibly infinite set of actions, where $\Lambda^?$ is the set of input actions, $\Lambda^!$ is the set of output actions, and Λ^τ is the set of internal actions;*

- *$\rightarrow \subseteq Q \times \Lambda \times Q$ is the transition relation.*

$\diamond$

Intuitively, the IOLTS semantics of an IOSTS $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ enumerates the possible values of the variables V and parameters P through the *valuations* of their domains. A *valuation* of the variables V is a mapping ν which maps every variable $x \in \mathcal{V}$ to a value $\nu(x)$ in the domain of x. Valuations of parameters P are defined similarly.

Let $\mathcal{V}$ denote the set of valuations of the variables V and let Γ denote the set of valuations of the parameters P. Considering $\nu \in \mathcal{V}$ and $\gamma \in \Gamma$, an expression E involving a subset of $V \cup P$, denoted by $E(\nu, \gamma)$, is the value obtained by evaluating the result of the replacement in E of each variable by ν and each parameter by γ.

Definition 2.3 (IOLTS semantics of an IOSTS)**.** *The semantics of an IOSTS $S = \langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ is an IOLTS $\mathcal{S} = \langle Q, Q^0, \Lambda, \rightarrow \rangle$, defined as follows [35]:*

- *$Q = L \times \mathcal{V}$ is the set of states;*
- *$Q^0 = \{\langle l^0, \nu \rangle \mid \Theta(\nu) = true\}$ is the set of initial states;*
- *$\Lambda = \{\langle a, \gamma \rangle \mid a \in \Sigma, \gamma \in \Gamma_{sig(a)}\}$ is the set of actions, where Λ is partitioned into the sets $\Lambda^?$ of input actions, $\Lambda^!$ of output actions, and Λ^τ of internal actions;*
- *$\rightarrow$ is the smallest relation in $Q \times \Lambda \times Q$ defined by the following rule:*

$$\frac{\langle l, \nu \rangle, \langle l', \nu' \rangle \in Q \quad \langle a, \gamma \rangle \in \Lambda \quad t : \langle l, a, G, A, l' \rangle \in \mathcal{T} \quad G(\nu, \gamma) = true \quad \nu' = A(\nu, \gamma)}{\langle l, \nu \rangle \overset{\langle a, \gamma \rangle}{\rightarrow} \langle l', \nu' \rangle}.$$

$\diamond$

Intuitively, the rule says that the system moves from a state $\langle l, \nu \rangle$ to a state $\langle l', \nu' \rangle$ through an action $\langle a, \gamma \rangle$ if there is a transition $t : \langle l, a, G, A, l' \rangle$ whose guard G is evaluated to $true$. Finally, the assignment A maps the pair (ν, γ) to ν'.

Next, we present some important definitions used to define the conformance relation between IOSTS specifications and the implementation of the system under test.

Definition 2.4 (Traces)**.** *Let L^0 denote the set of initial states. For an IOSTS $\mathcal{R}$ we denote by $traces(\mathcal{R})$ the set $\{\sigma \in (\Sigma^? \cup \Sigma^!)^* \mid \exists l^0 \in L^0, \exists l \in L, l^0 \overset{\sigma}{\Rightarrow} l\}$.* $\diamond$

Definition 2.5 (After)**.** *For $\sigma \in (\Sigma^? \cup \Sigma^!)^*$, we denote by $\mathcal{R}$ after σ the following set of states: $\{l \in L \mid \exists l^0 \in L^0, l^0 \overset{\sigma}{\Rightarrow} l\}$.* ◇

Definition 2.6 (Out)**.** *For $L' \subseteq L$ be a set of states, we denote by $out(L')$ the set of valued outputs that can be observed in states $l' \in L'$, that is, $out(L') = \{\alpha \in \Omega \mid \exists l' \in L', \exists l \in L, l' \overset{\alpha}{\Rightarrow} l\}$.* ◇

Definition 2.7 (Pref)**.** *For a set of traces T, we denote by pref(T) the set of strict prefixes of sequences in T.* ◇

Next, the formal framework for conformance testing presented in [118] (Subsection 2.1.5) and the formal framework for test purposes presented in [41] (Subsection 2.1.6) are instantiated. For this, the following concepts related to the frameworks must be defined: specifications, implementations, test purposes, test cases, verdicts, and the conformance relation.

Specifications. A specification is an IOSTS without cycles of internal actions.

Implementations. An implementation can be any computer system that can be modelled by an IOSTS.

Test Purposes. A test purpose is an IOSTS that describes a specific scenario to be verified. Before seeing the formal definition of a test purpose it is important to know two characteristics of IOSTS: completeness e compatibility.

Definition 2.8 (Completeness)**.** *An IOSTS is complete if for each $l \in L, \alpha \in \Sigma$, the set $\{l' | l \xrightarrow{\alpha} l'\}$ is non-empty, that is, each location allows all actions.* ◇

Definition 2.9 (Compatibility)**.** *Let $S_1 = \langle V_1, P_1, \Theta_1, L_1, l_1^0, \Sigma_1, \mathcal{T}_1 \rangle$ and $S_2 = \langle V_2, P_2, \Theta_2, L_2, l_2^0, \Sigma_2, \mathcal{T}_2 \rangle$ be two IOSTS. We say that S_1 and S_2 are compatible if $V_1 \cap V_2 = \emptyset$, $P_1 = P_2$, $\Sigma_1^! = \Sigma_2^!$, $\Sigma_1^? = \Sigma_2^?$, and $\Sigma_1^\tau \cap \Sigma_2^\tau = \emptyset$.* ◇

Definition 2.10 (Test Purpose)**.** *Let $S = \langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ be an IOSTS. A test purpose of S is an IOSTS $\mathcal{TP} = \langle V_{\mathcal{TP}}, P_{\mathcal{TP}}, \Theta_{\mathcal{TP}}, L_{\mathcal{TP}}, l^0_{\mathcal{TP}}, \Sigma_{\mathcal{TP}}, \mathcal{T}_{\mathcal{TP}} \rangle$ together with a set of locations $Accept_{\mathcal{TP}} \subseteq L_{\mathcal{TP}}$ and $Reject_{\mathcal{TP}} \subseteq L_{\mathcal{TP}}$ such that $\mathcal{TP}$ is complete and compatible with S.* ◇

Figure 2.4 presents an example of a test purpose for the triangle problem example. It is used to select scenarios where the user chooses inputs such that the triangle is equilateral.

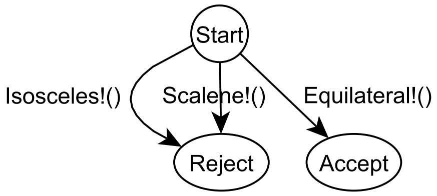


Figura 2.4: Test Purpose Example

The *Reject* location is used to discard all other scenarios where the system does not exhibit the desired behaviour. This test purpose is not complete but it is implicitly completed by the test case generation tool, so the activity of defining test purpose is simplified by allowing to focus only on the desired behaviour.

Test Cases. Test cases are used to assign verdicts to implementations.

Definition 2.11 (Test Case)**.** *A test case is an input-complete, deterministic IOSTS with three disjoints sets of locations:* Pass*,* Inconclusive*, and* Fail*.* ◇

An example of a test case is showed in Figure 2.5. It starts by providing three integer values to an implementation of the triangle problem example. Then it expects to receive a message informing that the chosen values represent the sides of a triangle. Next, if the system says that the triangle is equilateral, the verdict is **Pass**, that is, the implementation is in conformance with the specification and the test purpose. If some other response allowed

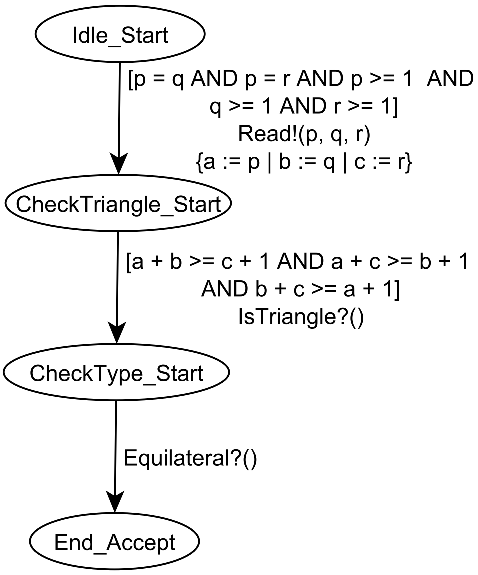

Idle_Start
[p = q AND p = r AND p >= 1 AND q >= 1 AND r >= 1]
Read!(p, q, r)
{a := p | b := q | c := r}
CheckTriangle_Start
[a + b >= c + 1 AND a + c >= b + 1 AND b + c >= a + 1]
IsTriangle?()
CheckType_Start
Equilateral?()
End_Accept

Figura 2.5: Test Case Example

by the specification is emitted then the verdict is **Inconclusive**. Finally, if an unspecified output is emitted then the verdict is **Fail**.

Conformance. The conformance relation links an implementation to the specification and the test purpose. In order to formally define the conformance relation it is needed to define a product operation that identifies in the specification all possible traces obtained with the specified test purpose.

Definition 2.12 (Product)**.** *The product $\mathcal{P} = \mathcal{S}_1 \times \mathcal{S}_2$ of two compatible IOSTS $\mathcal{S}_1$, $\mathcal{S}_2$ is the IOSTS $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ defined by: $V = V_1 \cup V_2$, $P = P_1 = P_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $L = L_1 \times L_2$, $l^0 = \langle l_1^0, l_2^0\rangle$, $\Sigma^? = \Sigma_1^? = \Sigma_2^?$, $\Sigma^! = \Sigma_1^! = \Sigma_2^!$, $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$. $\mathcal{T}$ is the smallest set of transitions satisfying the following rules [35]:*

1. $\frac{\langle l_1,a,G_1,A_1,l_1'\rangle \in \mathcal{T}_1,\ \ a \in \Sigma_1^\tau,\ \ l_2 \in L_2}{\langle\langle l_1,l_2\rangle,a,G_1,A_1\cup(x:=x)_{x\in V_2},\langle l_1',l_2\rangle\rangle \in \mathcal{T}}$ *and* $\frac{\langle l_2,a,G_2,A_2,l_2'\rangle \in \mathcal{T}_2,\ \ a \in \Sigma_2^\tau,\ \ l_1 \in L_1}{\langle\langle l_1,l_2\rangle,a,G_2,A_2\cup(x:=x)_{x\in V_1},\langle l_1,l_2'\rangle\rangle \in \mathcal{T}}$

2. $\frac{\langle l_1,a,G_1,A_1,l_1'\rangle \in \mathcal{T}_1\ \ \ \langle l_2,a,G_2,A_2,l_2'\rangle \in \mathcal{T}_2}{\langle\langle l_1,l_2\rangle,a,G_1\wedge G_2,A_1\cup A_2,\langle l_1',l_2'\rangle\rangle \in \mathcal{T}\ \ \ (for\ a \in \Sigma^! \cup \Sigma^?)}$

◇

The Rule 1, defined above, allows internal actions to evolve independently in each IOSTS and Rule 2 allows the synchronization of the observable actions of the two IOSTS.

Definition 2.13 (Atraces)**.** *Let $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$ be the product of the specification $\mathcal{S}$ with a test purpose $\mathcal{TP}$. Then $Atraces(\mathcal{P})$ is the set of traces of the specification that are selected according to the test purpose.* ◇

Definition 2.14 (Conformance Relation)**.** *Let $\mathcal{S}$ be a specification modelled by an IOSTS, $\mathcal{TP}$ be a test purpose for $\mathcal{S}$, and $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$ their product.*

1. *An implementation I is in conformance with the specification $\mathcal{S}$, denoted by $I\ conf\ \mathcal{S}$, if for all traces $\sigma \in traces(\mathcal{S}) : out(I\ after\ \sigma) \subseteq out(\mathcal{S}\ after\ \sigma)$.*

2. *An implementation I is in conformance with the specification $\mathcal{S}$ and the test purpose $\mathcal{TP}$, denoted by $I\ conf_{\mathcal{TP}}\ \mathcal{S}$, if for all traces $\sigma \in pref(Atraces(\mathcal{S} \times \mathcal{TP}))$: $out(I\ after\ \sigma) \subseteq out(\mathcal{S}\ after\ \sigma)$.*

◇

Intuitively, an implementation conforms to a specification if for all traces of the specification, the set of output actions of the implementation is contained in the set of output actions of the specification. In the second case, an implementation conforms to a specification and a test purpose if the same inclusion holds for each prefix of a trace of the product between specification and test purpose.

2.2 Real-Time Systems

At the same proportion in which computer systems become increasingly complex and ubiquitous in our lives, applications with time restrictions have become increasingly common. Among some examples of these types of applications we can cite military systems, control systems for chemical and nuclear industries, multimedia systems, air traffic control systems, elevator control systems, monitoring of patients in hospitals, embedded systems in robots, cars, airplanes, etc. All these applications have an important characteristic in common: time requirements. For such systems with explicit time requirements, the correct behaviour depends not only on the correctness of the results but also depends on the time at which they are produced [31]. Computer systems with this kind of restrictions are known as Real-Time Systems (RTS).

The classical view of computer systems is that, at some level of abstraction, they are regarded as a black box that receives inputs and provides appropriate outputs, finishing its execution after that (e.g., compilers, numerical analysis applications, and so on). However, most of current computer systems constantly interact with the environment around them continuously sending responses to input stimuli from the environment. These systems are characterized by their executions never finishing and are known as Reactive Systems. In general, RTS fit in the concept of Reactive Systems. In this context, we can say that RTS are computer systems that respond to input stimuli, originated from the environment in which they are inserted, in specific times [79].

According to the level of compliance with time requirements, real-time systems can be classified into Soft Real-Time Systems and Hard Real-Time Systems. Delays related to compliance with a response-time constraint (deadline) can be tolerated in a soft real-time system. On the other hand, systems in which failure to meet response-time constraints may lead to complete and catastrophic system failure are called hard real-time systems [79].

Soft real-time systems are typically used in a scenario where some elements concurrently interact to produce outputs due to input stimuli, for instance, some packets can be dropped in audio or video applications. In this case, violation of constraints results in degraded quality, but the system can continue to operate. A word-processing application is another example since it should respond to stimuli within a reasonable amount of time or its use will be impractical.

Hard real-time systems are used when it is essential to react to an event within a strict deadline. This kind of system requires strong time requirements because the loss of a deadline can mean losses in financial, environmental, or human terms. As examples of hard real-time systems we can cite medical systems such as heart pacemakers, chemical and nuclear industrial process controllers, a car engine control system, and so on. Hard real-time systems are typically found interacting at a low level with physical hardware, in embedded systems.

2.2.1 Modelling Time

In the study of real-time systems, one essential question is the nature of time. Specifying timing properties is difficult and may take different focuses. The focus considered in this work is that time can be classified into discrete or dense time [5].

The discrete or digital-time model considers time as being a monotonically increasing sequence of integers. This model allows to quantitatively express the distance between two events and establish total orders between them with a high level granularity. One of the advantages of this model is that its transformation to other formal languages is easier. One of the disadvantages is that the events of the real world do not always happen at integer-valued times.

In a dense or analogue-time model time increase monotonically as a sequence of real numbers. This model also allows quantitatively express the distance between two events but in a low level granularity. This model is more natural to represent events of the real world because everything happens in a continuous time. One disadvantage is that dense-time models are not simple to transform to a formal language and they are harder to analyse than in the discrete case.

2.2.2 Events

The nature of events is another important concept related to the study of real-time systems. Considering software systems in general, a change in state results in a change in the flow of control of the system. This change in the flow of control can be triggered by commands like if-then-else, case, invocation of procedures or methods, and so on. Thus, an event is any occurrence that causes a change in the flow of control of a software system [79].

Considering the context of real-time systems, events can be classified into synchronous and asynchronous events. The former are those that occur at predictable points in the flow of control and are represented by conditional branches, invocation of procedures or methods, occurrence of internal trap interruptions (in the case of exception handling), etc. The latter occur at unpredictable points in the flow of control. An important characteristic of the asynchronous events is that they are usually caused by external sources, for instance, an alarm system of a building has sensors to detect intruders and once a movement has been detected, the sensors interrupt the main application of the alarm system. In this scenario, the main application of the alarm system cannot predict when an event will occur because it is caused by external sources.

Synchronous and asynchronous events can be classified into periodic and aperiodic events [79]. Considering the alarm system example cited above, as the events do not occur at regular intervals they are called aperiodic asynchronous events. When interruptions are generated by a periodic external clock they can be classified as periodic asynchronous events. A periodic synchronous event is one represented by a sequence of invocation of tasks in a cyclic code, for instance a cyclic invocation of a method. A conditional branch that is not part of a code block (e.g. garbage collection) represents an aperiodic synchronous event.

2.2.3 Modelling Real-Time Systems

There are several formalisms in the literature for modelling real-time systems: timed automata [5], timed CSP [107], time Petri nets [16], real-time logics [22], and timed extended finite state machines [96]. This subsection introduces the most used formalisms: timed labelled transition systems and timed automata with their variations.

Timed Labelled Transition Systems (TLTS) is the simplest model (Definition 2.15). It

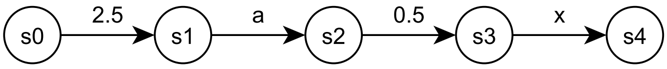


Figura 2.6: TLTS Example

is an extension of the classic LTS where the actions are divided into discrete and time-elapsing actions. The difference between discrete and time-elapsing actions is that the former occurs instantaneously, i.e. without consuming time, whereas the latter represents the time evolution.

Definition 2.15 (TLTS). *Formally, a TLTS is tuple $\langle S, s_0, Act, T\rangle$, where:*

- *S is a finite, non-empty set of states;*
- *$s_0 \in S$ is the initial state;*
- *$Act = A \cup D$ is a set of actions, where A is a finite set of discrete actions and $D = \{d \mid d \in \mathbb{R}^{\geq 0}\}$ is a set of time-elapsing actions;*
- *$T \subseteq S \times Act \times S$ is the transition relation with the following properties:*
  - ***Time Determinism:** $\forall s, s', s'' \in S$: if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$, then $s' = s''$*
  - ***Time Additivity:** $\forall s, s'' \in S$, $\forall d_1, d_2 \geq 0$: $(\exists s' \in S : s \xrightarrow{d_1} s' \xrightarrow{d_2} s'')$ iff $s \xrightarrow{d_1+d_2} s''$*
  - ***Null Delay:** $\forall s, s' \in S : s \xrightarrow{0} s'$ iff $s = s'$.*

◇

An example of a TLTS is shown in Figure 2.6. That example represents a scenario where, starting in state s_0, we delay for 2.5 units of time, after which we reach s_1, where we immediately execute action a, reach s_2, delay for 0.5 units of time, after which we reach s_3, where we immediately execute x, and reach state s_4.

The Timed Input-Output Labelled Transition System (TIOLTS) is an extension of TLTS where the set of discrete actions A is partitioned into input and output actions.

Definition 2.16 (TIOLTS). *A TIOLTS is a timed labelled transition system $\langle S, s_0, Act, T\rangle$ with $Act = A_I \cup A_O \cup D$ ($A_I \cap A_O = \emptyset$), where A_I is a finite set of input actions and A_O is a finite set of output actions.* ◇

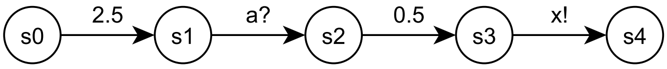


Figura 2.7: TIOLTS Example

Figure 2.7 presents an example of TIOLTS. It means that, starting in state s_0, we delay for 2.5 units of time, after which we reach s_1, where we immediately provide the input action a to the system, reach s_2, delay for 0.5 units of time, after which we reach s_3, where the system immediately responds with the output action x, and reach state s_4.

Most of the work related to real-time model checking and testing is based on Timed Automata (TA). It was firstly proposed by Alur and Dill [5] and ever since many variations have been proposed.

Definition 2.17 (TA). *A Timed Automaton is a tuple $\langle Q, q_0, Act, C, E\rangle$, where:*

- *Q is a finite set of locations;*
- *$q_0 \in Q$ is the initial location;*
- *Act is a finite set of actions;*
- *C is a finite set of clocks;*
- *E is a finite set of transitions. Each transition is a tuple $\langle q, q', a, \lambda, \delta\rangle$, where:*
  - *$q, q' \in Q$ are the source and destination locations,*
  - *$a \in Act$ is the action of the transition,*
  - *$\lambda \subseteq C$ is the set of clocks to reset to zero,*
  - *δ is a clock constraint over C. δ is defined inductively by*

    $$\delta := x\#c \mid \delta_1 \wedge \delta_2,$$

    where c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$.

$\diamond$

Figure 2.8 presents an example of a timed automaton that models the same scenario represented in Figure 2.6. Considering timed automata, it is assumed all clocks are reset to

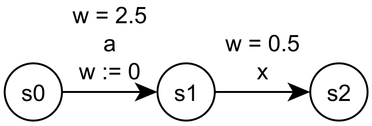


Figura 2.8: TA Example

zero at the beginning. In Figure 2.8, there is a single clock w and the notation $w := 0$ means the action of resetting the clock w. Similarly, the notations $w = 2.5$ and $w = 0.5$ represent the clock constraints associated with the transitions. The automaton starts in location s_0 and, once the transition is enabled (i.e. $w = 2.5$), the action a is executed, reaching the location s_1. The clock is reset to zero along with this transition. The time elapsed since the occurrence of the action a is shown by the value of clock w. The transition from location s_1 to location s_2 is enabled only if this value is equal to 0.5, where the action x is immediately executed.

The semantics of TA can be defined in terms of an infinite TLTS (Definition 2.18). Let the function $\upsilon : C \to \mathbb{R}^{\geq 0}$ denote a clock valuation υ.

Definition 2.18 (TLTS semantics of a TA)**.** *The semantics of a TA $A = \langle Q, q_0, Act, C, E\rangle$ is a TLTS $L_A = \langle S, s_0, Act, T\rangle$, defined as follows:*

- *$S = Q \times (C \to \mathbb{R}^{\geq 0})$ is the set of states of the form $s = (q, \upsilon)$ where $q \in Q$ is a location and υ is a clock valuation;*
- *$s_0 = (q_0, \vec{0})$ is the initial state, where $\vec{0}$ is the valuation assigning 0 to every clock in C;*
- *$Act = A \cup D$ is the set of actions, where A is the set of discrete actions and $D = \{d \mid d \in \mathbb{R}^{\geq 0}\}$ is the set of time-elapsing actions;*
- *T is the transition relation defined as follows: (1) transitions with discrete actions are of the form $(q, \upsilon) \xrightarrow{a} (q', \upsilon')$, where $a \in Act$ and there is a transition $\langle q, q', a, \lambda, \delta\rangle$, such that υ satisfies δ and υ' is obtained by resetting to zero all clocks in λ; (2) transitions with time-elapsing actions are of the form $(q, \upsilon) \xrightarrow{d} (q, \upsilon + d)$ for all $d \in \mathbb{R}^{\geq 0}$ such that $\upsilon \models \delta$ and $\upsilon + d \models \delta$.*

◇

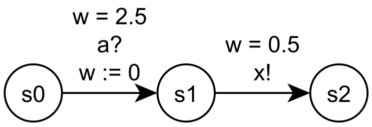


Figura 2.9: TAIO Example

Timed Automata with Inputs and Outputs (TAIO) is an extension of TA where the set of actions Act is partitioned in two disjoint sets: a set of input actions and a set of output actions.

Definition 2.19 (TAIO)**.** *A TAIO is a timed automata $\langle Q, q_0, Act, C, E\rangle$ where $Act = Act_I \cup Act_O$, such that Act_I is a finite set of input actions and Act_O is a finite set of output actions. Moreover, $Act_I \cap Act_O = \emptyset$.* ◇

The semantics of TAIO can be defined in terms of an infinite TIOLTS in a similar manner as described for TA. Figure 2.9 presents an example of a TAIO. This example represents the scenario of Figure 2.8 and it states that the system must receive the input a exactly with 2.5 units of time. Finally, the system must output b exactly with half of one unit of time.

As the semantics of TA and TAIO is defined in terms of infinite timed labelled transition systems, both verification and testing techniques must deal with large sets of states which may lead to the state space explosion problem. In this case, it is important to have an efficient symbolic representation of the state space. One of the most efficient representations is based on the notion of zone [42; 58; 122; 123]. A zone represents the maximal set of clock valuations that satisfy a constraint.

The analysis of the state space using this symbolic representation requires some operations such as:

- The future of a zone Z, defined by $\vec{Z} = \{z + d \mid z \in Z,\ d \in \mathbb{R}^{\geq 0}\}$;
- The intersection of two zones Z and Z', defined by $Z \cap Z' = \{z \mid z \in Z,\ z \in Z'\}$;
- The reset to zero $\lambda \subseteq C$ of Z, defined by $[\lambda \leftarrow 0]Z = \{[\lambda \leftarrow o]z \mid z \in Z\}$.

All these operations are graphically illustrated in Figures 2.10, 2.11, and 2.12, respectively.

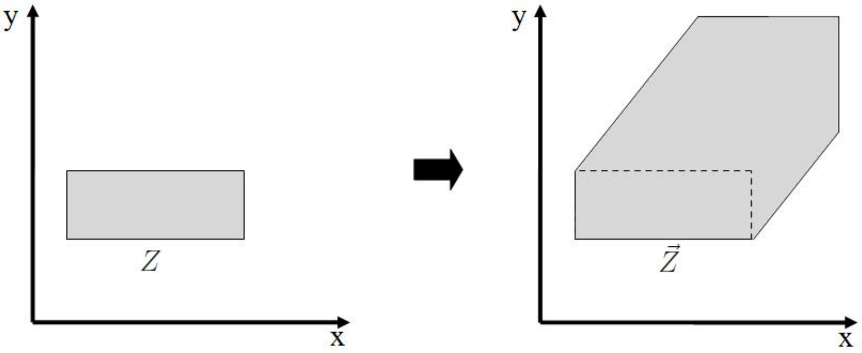


Figura 2.10: Example of the Future Operation

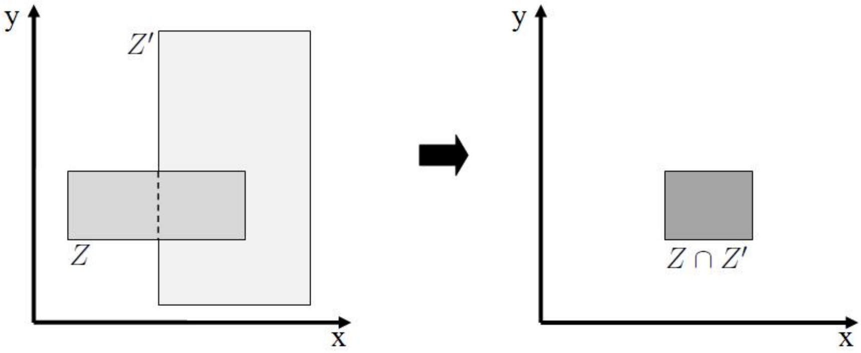


Figura 2.11: Example of the Intersection Operation

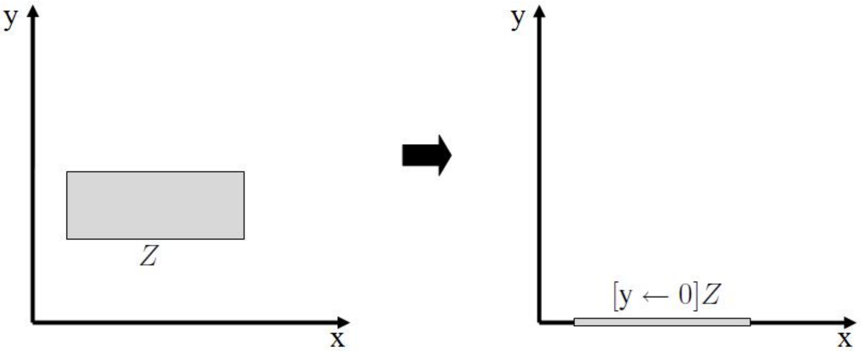


Figura 2.12: Example of the Reset to Zero Operation

2.2.4 Testing of Real-Time Systems

The increasing use of real-time systems, in most different contexts, has been demanding investments in order to increase the reliability and integrity of such systems. Some research efforts have been expended in devising techniques such as model checking [13; 31; 54], where the correctness of models is verified in an automated and accurate manner. However, if the same rigour is not applied to the test of the implementation, a gap is created between these processes, allowing the presence of defects in the implementation even if the model had been successfully verified.

Since research in the real-time software testing field is very recent, the developed techniques and tools are still immature and difficult to use in practice. Thus, one of the challenges today is the search for methods, techniques and tools to support the test of systems with time restrictions. Nevertheless, real-time systems have several distinguishing characteristics that may need to be taken into account during the testing process, leading to the most difficult challenges in software testing [121].

The test of real-time systems is more difficult than the test of non-real-time systems because the correct behaviour of the former depends not only on the correct results but also when they are emitted. Thus, it is essential to develop techniques to interact with the system: (1) by providing inputs at the correct time and (2) by observing and evaluating if the generated results are correct in terms of integrity and timing.

Real-time systems are usually composed of parallel activities. So, the models must represent such parallelism between many elements and allow ways of communication between these elements. As previously said, RTS are extremely related to events that often occur in terms of interruption signals from the arrival of data, ticking of a hardware clock, or an error alarm. To provide an effective solution for testing, it is crucial to define models capable of representing these asynchronous events. In addition, the model has to be composable, allowing events to be combined at different points of possibly different flows of execution. Research in this direction is practically nonexistent and some approaches only consider interruptions in a non-real-time context [7; 8; 9; 12; 29; 39].

Another hard activity of testing real-time systems is the test execution. Considering an environment composed of several processes executing at the same time with synchronous and asynchronous events, it is very difficult to have control of the whole environment. Current work in the literature consider so many hypotheses related to the environment and the system under test that the results sometimes are not useful in practice. It is needed testing techniques and theories that allow one to have interesting conclusions without having the environment fully controllable.

A pass verdict is emitted when the system produces an acceptable result on time. The addition of time in the validation of RTS, the unpredictability of events, and the absence of controllability during the test execution lead to a harder oracle problem. This happens due to the fact that it is very difficult to have total control of an environment with parallel activities during the test execution. Moreover, in practice, the time cannot be controlled and it is treated in different abstraction levels by the tester and the specification. In this case, test hypotheses must be defined in order to achieve valid verdicts. To the best of our knowledge, approaches in the direction of solving the oracle problem considering the context of real-time systems are practically nonexistent.

2.3 Concluding Remarks

This chapter presented the theoretical basis necessary for the understanding of this work. At the beginning, the main concepts related to software testing were presented, for instance test cases, oracles, approaches to identify test cases, conformance testing, and testing techniques like model-based testing, property oriented testing, and symbolic testing.

With respect to the real-time systems context, we described what a real-time system is and discussed several characteristics like how the time can be modelled, the types of existing events, some classical notations used to model RTS, and the difficulties to test real-time systems.

Capítulo 3

Teste de Interrupção em Sistemas Reativos

Para prover uma solução efetiva para o teste de interrupção é crucial definir um modelo de teste capaz de representar tais interrupções e, consequentemente, tornar possível a geração automática de casos de teste. Tal modelo de teste deve ser passível de composição, permitindo a especificação de interrupções em diferentes pontos de eventualmente diferentes fluxos de execução. Assim, devido à grande quantidade de casos de teste possíveis, estratégias de seleção precisam ser aplicadas para reduzir o tamanho das suítes de teste. Além disso, o ambiente de execução de teste deve ser cuidadosamente considerado para que os requisitos e as restrições de execução sejam devidamente identificados e tratados.

Este capítulo apresenta uma abordagem de teste de conformidade para sistemas reativos com interrupções, que abrange a modelagem (voltada para teste), geração e seleção de casos de teste consistentes [9]. O modelo adotado é denominado *Annotated Labelled Transition System* (ALTS). Este tipo de *Labelled Transition System* (LTS) tem descrições especiais inseridas no modelo, a fim de tornar o processo de geração de casos de teste possível. LTSs são bons modelos para testes funcionais, pois todas as informações necessárias são as interações observáveis entre as aplicações e o ambiente e também entre as próprias aplicações. Além disso, eles são o formalismo base da maioria das notações formais para aplicações reativas. O modelo proposto está implementado na ferramenta LTS-BT [29] e um estudo de caso é realizado para ilustrar os benefícios da abordagem em relação à seleção manual.

Teste de interrupção foi investigado em [6] considerando o contexto das aplicações para celulares, mas apenas uma estratégia operacional foi proposta. Este capítulo apresenta uma abordagem formal para sistemas reativos em geral. Além disso, o trabalho apresentado em [9] e descrito neste capítulo estende o trabalho apresentado em [8], abrangendo os seguintes aspectos: (1) algoritmos definidos para a tradução de especificações em alto nível para modelos são apresentados; (2) o estudo de caso é apresentado em mais detalhes e os resultados são amplamente discutidos.

3.1 Context

In general, the test process in the context of this chapter starts with a specification of the System Under Test (SUT) and interruptions. Given high level specifications, an ALTS model is automatically generated. Finally, the ALTS model is combined with test purposes for interruption test case generation. The interruption test process uses test purposes in order to test at specific points of interest. A general view of this test process is presented in Figure 3.1. This process considers the test architecture presented in Figure 3.2. In this test architecture, two elements are important: the SUT and the TESTER. The SUT is composed of the main application and interruptions allowed during the test process. The environment is assumed to be fully controllable by the TESTER, thus, during test execution the TESTER has total control of the interruptions, deciding when they start and finish.

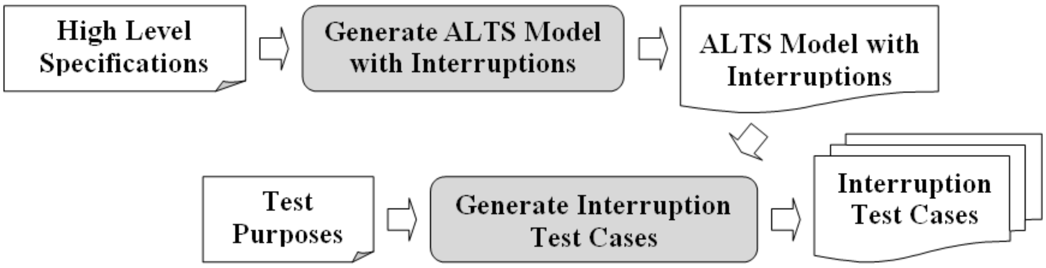

High Level Specifications
Generate ALTS Model with Interruptions
ALTS Model with Interruptions
Test Purposes
Generate Interruption Test Cases
Interruption Test Cases

Figura 3.1: Interruption Test Process

The SUT is specified as use cases using a controlled natural language [26; 84; 115]. An example of a use case of a mobile phone application is shown in Figure 3.3. This represents the behaviour of removing a message from inbox. A use case must have a main flow and can have some alternative flows. The flows are described through steps that include a user action and the respective system response. For instance, the step “4M"has the selection of

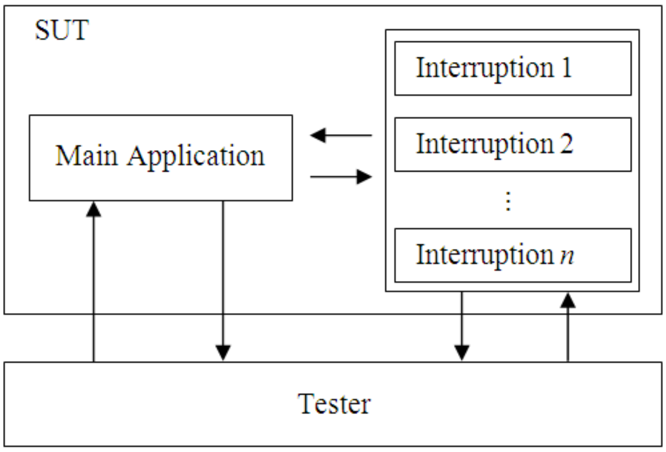


Figura 3.2: Test Architecture

the “Remove"option, and the respective system response is to show an alert saying that the message was removed.

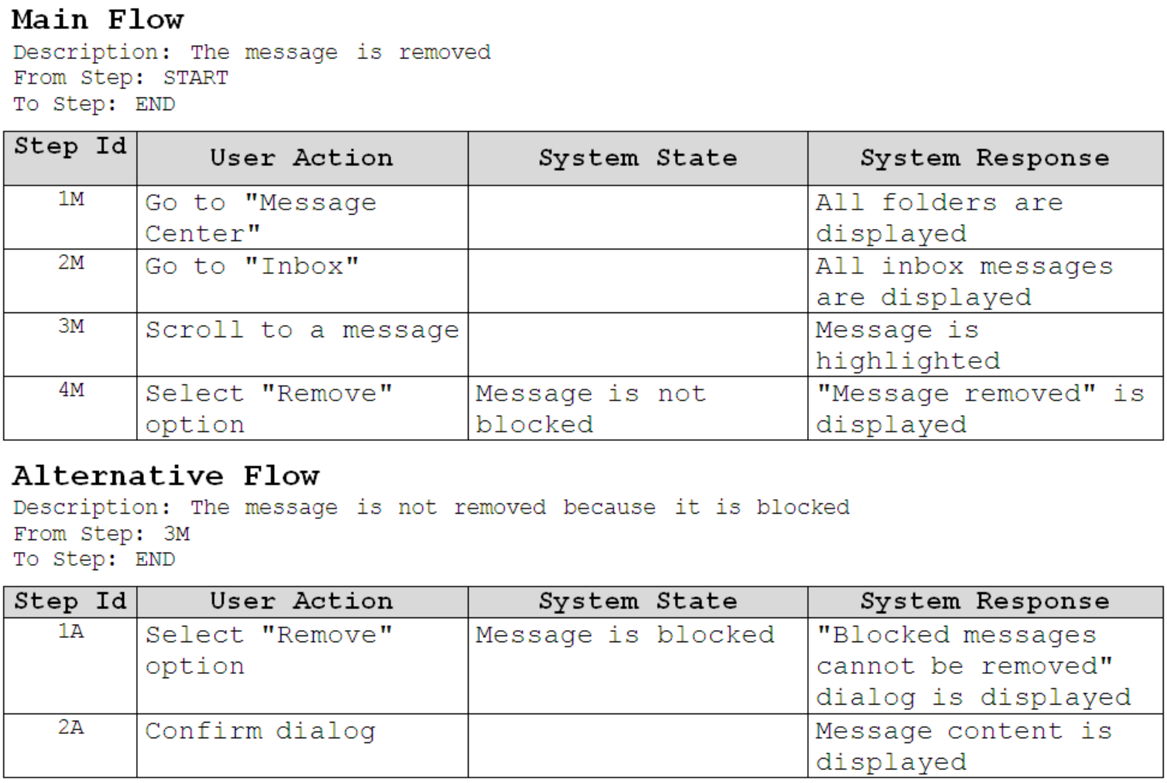
Besides the actor action and the system response, each step has a condition (System State) that determines if the system response will happen or not. If the condition is not satisfied, an alternative flow must be specified. As an example, the step “4M"of the main flow has one alternative flow (steps “1A"and “2A").

Considering the specifications of interruptions, the idea is to specify an interruption in the same way by using the same use case template that is used to specify a simple behaviour of the SUT [39]. For instance, Figure 3.4 presents the behaviour of an incoming alert interruption. This interruption specifies the arrival of a new kind of text messages where the text appears to the user inside a dialog box.

Once the interruption flow is specified, we assume that it can be executed at any time of another use case execution, that is, between any step of another use case. With this specification strategy, interruption behaviours are defined in a simple manner and all points where an interruption can occur do not need to be explicitly specified.

3.2 Interruption Model

This section presents the proposed ALTS model structure capable of representing interruptions. Firstly, interruptions are represented with IOLTS models (Definition 2.2) to illustrate the challenges and the desired semantics for ALTS models. Secondly, ALTS models are de-



Main Flow

Description: The message is removed
From Step: START
To Step: END

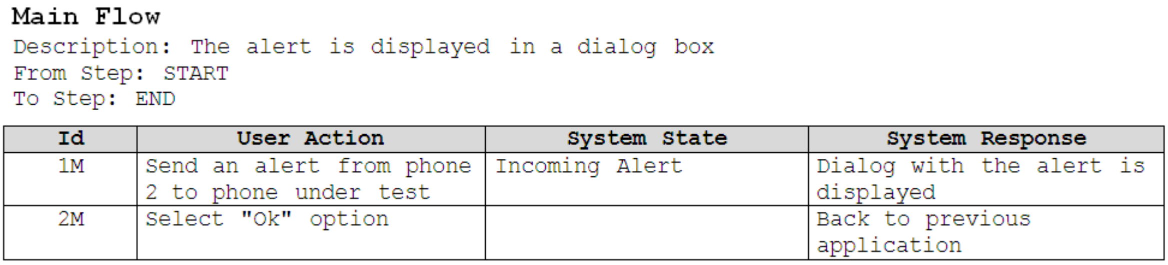
Step Id	User Action	System State	System Response
1M	Go to "Message Center"		All folders are displayed
2M	Go to "Inbox"		All inbox messages are displayed
3M	Scroll to a message		Message is highlighted
4M	Select "Remove" option	Message is not blocked	"Message removed" is displayed

Alternative Flow

Description: The message is not removed because it is blocked
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Select "Remove" option	Message is blocked	"Blocked messages cannot be removed" dialog is displayed
2A	Confirm dialog		Message content is displayed

Figura 3.3: *Remove Message* Specification



Main Flow

Description: The alert is displayed in a dialog box
From Step: START
To Step: END

Id	User Action	System State	System Response
1M	Send an alert from phone 2 to phone under test	Incoming Alert	Dialog with the alert is displayed
2M	Select "Ok" option		Back to previous application

Figura 3.4: *Incoming Alert Interruption* Specification

fined, their structure is illustrated by an application in the mobile phone domain, and their semantics are defined showing how an ALTS can be converted into an IOLTS. Finally, the notion of conformance considered is discussed.

3.2.1 Representing Interruptions with IOLTS Models

Considering the conformance testing approach, LTS is one of the most used formalisms. Basically, LTS models are represented by graphs where the nodes are the possible system states and the edges represent the transition between these states through occurrence of actions. LTSs can be used for modelling the behaviour of systems such as specifications, implementations, and tests, and it serves as a semantic model for several formal languages such as CCS and CSP [116; 117].

Particularly in the case of reactive systems, the underlying model should represent the interaction of the system with its environment by distinguishing between inputs and outputs. In this case, IOLTS models are used. Figure 3.5 shows an example of an IOLTS. An input event is defined using the symbol "?" followed by the event name and an output event is defined using the symbol "!" followed by the event name.

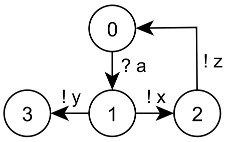


Figura 3.5: Simple IOLTS

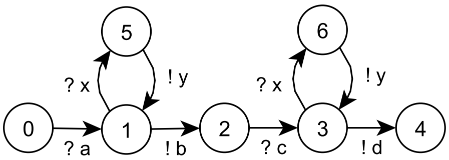


Figura 3.6: Modelling Interruptions Using IOLTS

It is possible to model interruptions using an IOLTS. For this, each possibility of interruption needs to have a specific set of states, implying that interruption flows must be duplicated. Figure 3.6 shows an example of how to model interruptions using IOLTS. Nodes from 0 to 4 are related to a behaviour that can be interrupted by another behaviour at nodes 1 and 3. State 5 represents the possibility of interruption at node 1 and state 6 the possibility of interruption at node 3. Note that nodes 5 and 6 represent the same interruption behaviour.

The replication of the interruption model is due to the semantics of the behaviour. Suppose that only one state had been used to represent the interruption behaviour, then it would not be possible to associate a unique next state to the end of the interruption execution. After an interruption execution, the flow needs to continue from the same point where the interruption had started.

3.2.2 Annotated Labelled Transition Systems

ALTSs are capable of representing interruptions in a more compact way, following the same semantics presented in the previous subsection. This new kind of LTS follows the same classical LTS definition. The difference is that each label is associated with a description. This new description inserted into the model is called an annotation. Before defining an ALTS, a definition of a Generic Annotated Labelled Transition Systems (GALTS) is presented.

Definition 3.1 (GALTS)**.** *A GALTS is a 5-tuple $\langle Q, A, L, q_0, T\rangle$, where:*

- *Q is a countable, non-empty set of locations;*
- *A is a countable, non-empty set of annotations;*
- *L is a countable, non-empty set of labels;*
- *$q_0 \in Q$ is an initial location;*
- *T is a set of transitions. Each transition consists of:*
  - *a location $q \in Q$, called the origin of the transition;*
  - *an annotation $a \in A$, called the annotation of the transition;*
  - *a label $l \in L$, called the label of the transition;*
  - *a location $q' \in Q$, called the destination of the transition.*

◇

As said before, each label has an associated description (annotation). So in the GALTS definition (Definition 3.1) we have a set A that contains the possible descriptions of the labels. This set can be instantiated according to the information to be modelled or the context where the model will be used. In this work, the focus is on a model to support the test process, mainly a model capable of representing interruptions efficiently. Thus, a more specific GALTS is defined where the set A of annotations has predefined elements.

Definition 3.2 (ALTS). *An ALTS is a 5-tuple $\langle Q, A, L, q_0, T\rangle$, where:*

- *Q is a countable, non-empty set of locations;*
- *A = {steps, conditions, expectedResults, beginInterruption_X, endInterruption_X} is the set of annotations;*
- *L is a countable, non-empty set of labels;*
- *$q_0 \in Q$ is an initial location;*
- *T is a set of transitions. Each transition consists of:*
  - *a location $q \in Q$, called the origin of the transition;*
  - *an annotation $a \in A$, called the annotation of the transition;*
  - *a label $l \in L$, called the label of the transition;*
  - *a location $q' \in Q$, called the destination of the transition.*

◇

These annotations were chosen with the following specific goals: (1) guide the test case generation process, by making the focus on particular interruptions easier; (2) make it possible for interruption models to be plugged and unplugged without interfering with the main model; (3) guide test case documentation; (4) make it possible for conditions to be associated with actions; (5) indicate points where interruptions can be reasonably observed externally.

The annotation *steps* is associated with a label $l \in L$ (we write $[steps]l$) to indicate that l is an input action. When a label $l \in L$ represents a condition associated with an input action, we use the annotation *conditions* and write $[conditions]l$. The expected results are indicated through *expectedResults* annotation ($[expectedResults]l$). Two other annotations are used to indicate the start and the end of an interruption and they are considered as special kinds of input actions and expected results, respectively. So the labels in *L* represent the observable actions (input or output actions) or some condition associated with these actions.

Let $W = \langle Q, A, L, q_0, T\rangle$ be an ALTS. We write $q \overset{[a]l}{\rightarrow} q'$ for $(q, a, l, q') \in T$ and $q \overset{[a]l}{\rightarrow}$ for $\exists q' : q \overset{[a]l}{\rightarrow} q'$. An ALTS can be defined by its initial location, then we write W → for $q_0 \rightarrow$. Depending on the associated annotation, the labels can be classified as input actions, output actions, and conditions. Thus, let $L = L_I \cup L_O \cup L_C$, where L_I is the set of input actions, L_O is the set of output actions, and L_C is the set of conditions. Let $a_{(i)} \in A$ be some annotations, $\omega_{(i)} \in L$ be some labels, $\sigma \in ([A]L)^*$ a sequence of labels with their respective annotations, and q, $q' \in Q$ some locations.

Let $\Omega(q) \triangleq \{[a]\omega \mid a \in A, \omega \in L, q \overset{[a]\omega}{\rightarrow}\}$ be the set of actions reachable from q. Also, let $Out(q) \triangleq \Omega(q) \cap [A \setminus \{steps, conditions, beginInterruption\_X\}]L_O$ be the set of outputs reachable from q. The definition of $Out(q)$ can be extended for sets of locations: for $P \subseteq Q$ we have $Out(P) \triangleq \bigcup_{q \in P} Out(q)$. Denote $q \overset{[a_1]\omega_1 \ldots [a_n]\omega_n}{\rightarrow} q' \triangleq \exists q_0, \ldots, q_n : q = q_0 \overset{[a_1]\omega_1}{\rightarrow} q_1 \overset{[a_2]\omega_2}{\rightarrow} \ldots \overset{[a_n]\omega_n}{\rightarrow} q_n = q'$. The set q *after* $\sigma \triangleq \{q' \in Q \mid q \overset{\sigma}{\rightarrow} q'\}$ is the set of locations reachable from q, and P *after* $\sigma \triangleq \bigcup_{q \in P} q$ *after* σ is the set of locations reachable from the set P. *Traces*$(q) \triangleq \{\sigma \in ([A]L)^* \mid q \overset{\sigma}{\rightarrow}\}$ describes the sequences of labels with their respective annotations reachable from q. Considering the sequences of labels and annotations reachable from the initial location of an ALTS W, we define *Traces*$(W) \triangleq$ *Traces*(q_0).

Considering our running example presented in Section 3.1, Figure 3.7 presents an ALTS model that represents the behaviour of removing a message from inbox. This application is specified by the use case shown in Figure 3.3. The model where interruptions can occur will be illustrated with the scenario where that feature specified by the use case shown in Figure 3.3 can be interrupted at some points by the Incoming Alert interruption (interruption specified by the use case presented in Figure 3.4). The scenario described above is presented in Figure 3.8. Note that locations from 0 to 13 are related to the remove message behaviour (Figure 3.3), and locations from 14 to 17 are related to the Incoming Alert interruption (Figure 3.4).

From Figure 3.8, the interruption model is connected to the feature that can be interrupted (the main flow) using two new annotations: *beginInterruption_X* and *endInterruption_X*, where *X* is a counter. These annotations are used to indicate where the main flow has been interrupted. Also, they are needed to represent the behaviour where the main flow continues its execution from the same point where it had been interrupted. For instance, if an interruption begins with the *beginInterruption_0* annotation it must finish with *endInterruption_0.*

One of the main advantages of using the Annotated LTS is that we can add the same interruption behaviour to many different points only by manipulating the two new annotations (*beginInterruption_X* and *endInterruption_X*). Thus, we can represent interruptions in a more compact way than standard LTS, while preserving the same efficiency and precision in test generation (this is discussed in Section 3.4).

Considering time as being continuous, an interruption can occur at infinite points during the system execution. But considering the tester's point of view, each possibility of interruption can only be observed after each system response. This happens due to the fact that it is impractical to reproduce a scenario where an interruption occurs between an input action and the system response, mainly when tests are manually executed. It is important to remark that this is a limitation of the test process in general and not of the proposal presented in this chapter. Thus, the intention is to represent only interruptions that occur immediately after the system responses. In this case, Figure 3.8 represents all possibilities of interruption from the tester's point of view.

Note that, as we are considering an LTS model for testing, only functionalities to be tested are specified. Thus, we have a partial behavioural model. From the tester's point of view, only the specified behaviour is observed, and with this, all other behaviours are not observed during the test, including other possible interruptions. We are assuming that the test execution environment is controlled by the tester, that is, an interruption only occurs when the tester wishes that it occur.

In practice, this interruption model should not be written by hand because it is tiresome and not cost-effective. It must be generated directly from abstract specifications. The ALTS

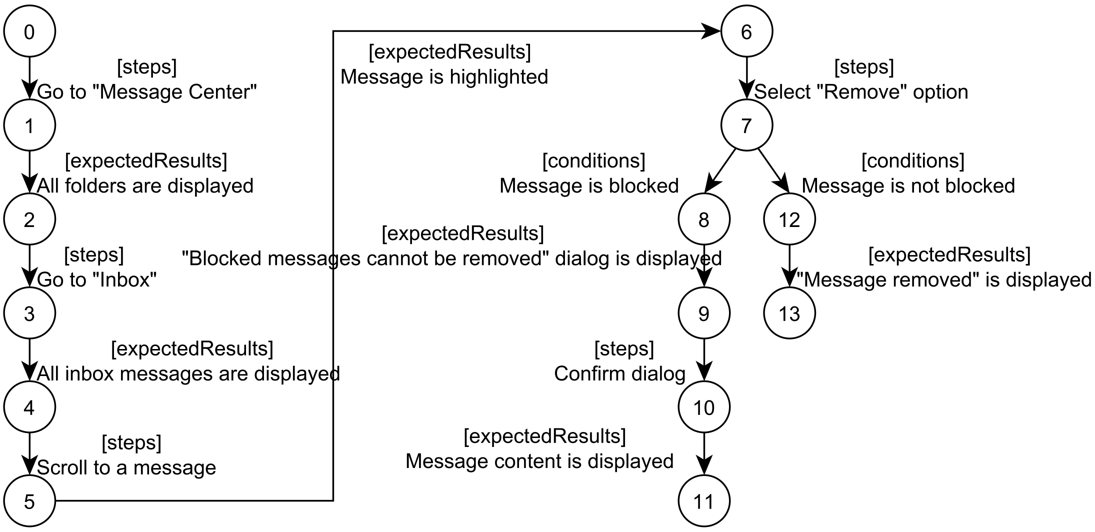


Figura 3.7: *Remove Message* behaviour

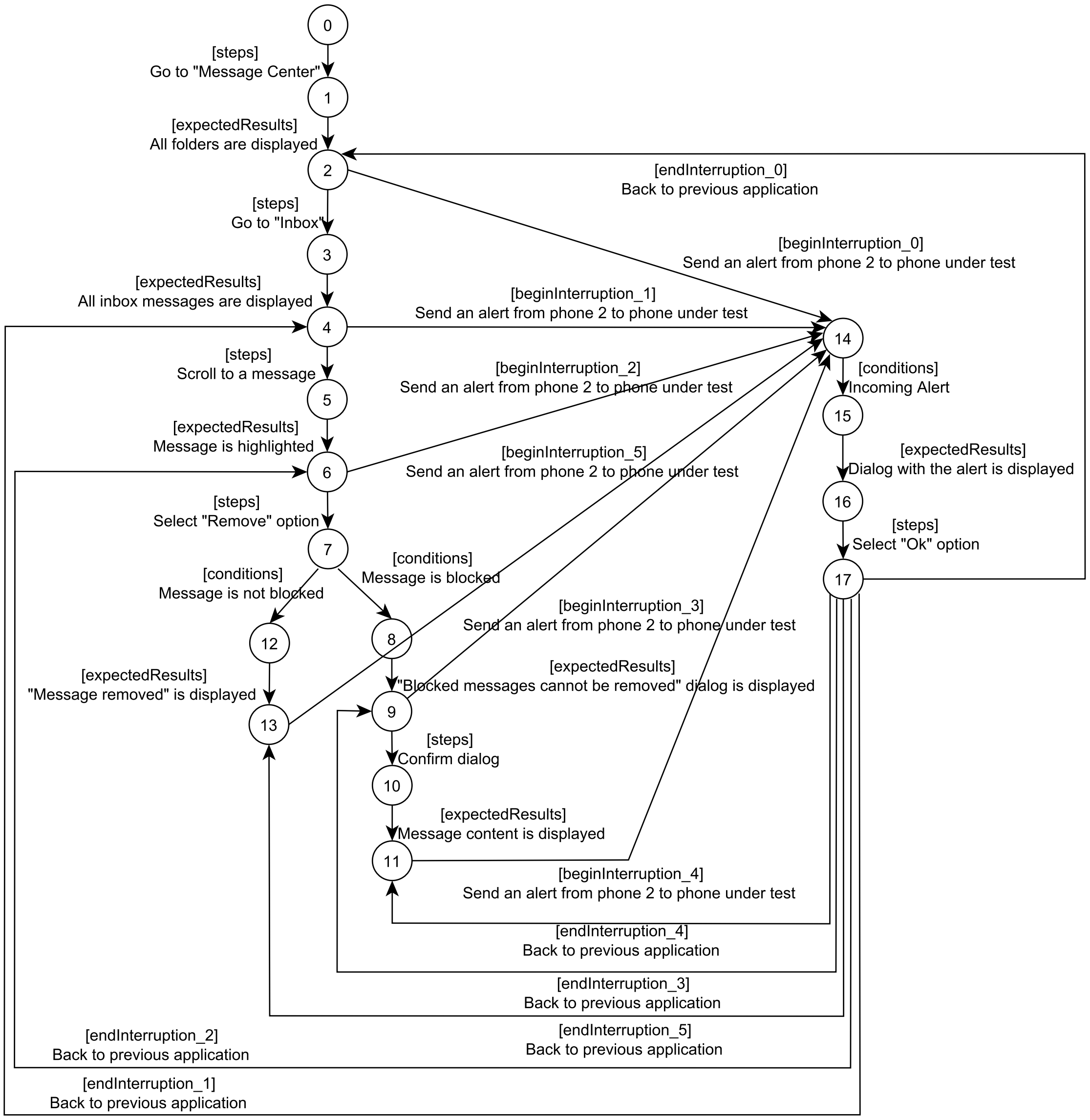

0
[steps]
Go to "Message Center"
1
[expectedResults]
All folders are displayed
2
[steps]
Go to "Inbox"
3
[expectedResults]
All inbox messages are displayed
4
[steps]
Scroll to a message
5
[expectedResults]
Message is highlighted
6
[steps]
Select "Remove" option
7
[conditions]
Message is not blocked
[conditions]
Message is blocked
12
8
[expectedResults]
"Message removed" is displayed
13
[expectedResults]
"Blocked messages cannot be removed" dialog is displayed
9
[steps]
Confirm dialog
10
[expectedResults]
Message content is displayed
11
[endInterruption_0]
Back to previous application
[beginInterruption_0]
Send an alert from phone 2 to phone under test
[beginInterruption_1]
Send an alert from phone 2 to phone under test
[beginInterruption_2]
Send an alert from phone 2 to phone under test
[beginInterruption_5]
Send an alert from phone 2 to phone under test
[beginInterruption_3]
Send an alert from phone 2 to phone under test
[beginInterruption_4]
Send an alert from phone 2 to phone under test
14
[conditions]
Incoming Alert
15
[expectedResults]
Dialog with the alert is displayed
16
[steps]
Select "Ok" option
17
[endInterruption_4]
Back to previous application
[endInterruption_3]
Back to previous application
[endInterruption_5]
Back to previous application
[endInterruption_2]
Back to previous application
[endInterruption_1]
Back to previous application

Figura 3.8: *Remove Message* behaviour with Interruptions

model presented in this section is automatically generated from those use case templates described in Section 3.1 by the LTS-BT tool [29]. This tool is described in more details in the next section.

The semantics of ALTS models can be defined in terms of IOLTS models. Basically, locations in the ALTS model are states in the IOLTS plus additional states that are created to replicate the interruption behaviour. Labels annotated with *steps*, *conditions*, *beginInterruption_X* are input actions in the IOLTS, whereas labels annotated with *expectedResults* and *endInterruption_X* are output actions in the IOLTS. The transition function is incremented by the replication of interruption behaviour by considering the new states added. The most non-trivial element in this association is the condition transition. The reason to map them to input action is that they often represent in test models different paths of execution of the application. When a condition is associated with a test case, this often means that the tester will need to properly set up the application so that a particular flow of execution can be tested. Therefore, in test case documentation, they are often promoted to an initial condition that will demand an input set up information.

3.2.3 Testing Conformance

This work considers a testing theory that is based on the notions of *specification*, *implementation*, and a *conformance relation* between them [118]. The specification of a reactive system with interruption can be written in any notation that can be transformed into an ALTS model. But this work considers only use case templates or an ALTS that respects the constraints on the use of labels defined in Subsection 3.2.2. The implementation can be any computer system that can be interrupted at any time and can be modelled as an ALTS. Moreover, it is assumed all interruptions to be controllable and implementations to be input-enabled, that is,

$$\forall q \in Q, a \in A \setminus \{conditions, expectedResults, endInterruption\_X\}, \forall \omega \in L_I, q \overset{[a]\omega}{\rightarrow} .$$

As discussed in Section 3.1, the tester needs full control of the test environment in order to achieve valid verdicts during the test execution process. Thus, the implementation under test must respond to all stimuli of the tester leading to require, at least during the test execution, that the implementation is input-enabled. This is a usual assumption of testing techniques (e.g. [62; 116; 117]) that can be achieved by appropriate control mechanisms. This work considers a conformance relation based on a simplification of the **ioco** relation defined by Tretmans in [117]. The **conf** relation (presented in Definition 3.3) does not take internal actions and quiescence into account.

Definition 3.3 (conf)**.** *Let the specification S be an ALTS and SUT be an input-enabled ALTS:*

$$\mathit{SUT}\ \textbf{\textit{conf}}\ S \triangleq \forall \sigma \in \text{Traces}(S),\ Out(\mathit{SUT} \text{ after } \sigma) \subseteq Out(S \text{ after } \sigma).$$

◇

It is important to mention that traces of ALTSs are restricted to paths in which the returns from interruptions go to the correct locations according to the ALTS semantics presented in the end of Section 3.2.2. Informally, an implementation conforms to a specification for **conf** if for all traces of the specification, the set of output actions of the implementation in each location is contained in the set of output actions of the specification. This implementation relation is similar to the one considered by the TGV tool [62].

3.3 Interruption Test Case Generation and Selection

This section presents the algorithms developed in order to automate the test process described in Section 3.1. Firstly, algorithms that translate use case templates to ALTS models are presented. After that, an algorithm that generates interruption test cases is shown. Finally, an interruption test case selection strategy based on test purposes is presented.

3.3.1 ALTS Model Generation from Use Case Templates

This subsection presents a strategy for translating use case templates into ALTS models from which test cases can be generated. The general translation procedure is shown in Algorithm 3.1. This procedure is a variation of the one presented in [93] that focus on individual features only. Basically:

- Each template of the use case, starting from the main flow one, is processed sequentially and, from each step, locations and transitions are created in the target ALTS according to the order of steps defined in the template. This is controlled by the two *for* loops;

- *currentLocation* represents the location from which transitions are created for the current step. This is either: (1) the last location created in case the *From Step* field is defined as *START* or this is the first location; or (2) the last location of a given step (defined in the *From Step* field) of another template;

- *From Step* and *To Step* guide the connection of each trace created by each of the templates;

- *User Action*, *System State*, and *System Response* become transitions that are associated with *steps*, *conditions*, and *expectedResults* annotations, respectively;

- Locations are created as new transitions that need to be added. These are incrementally numbered from 0. Locations and transitions are created by the add operation. But locations already created can be reused when connecting the traces of new templates. When this is possible, the *addToStep* (*To Step* is different from *END*) and *addFromStep* (*From Step* is different from *START*) are used instead;

- Duplicated transitions from the same location are also avoided. This can happen when the same steps are possible but with different conditions.

Algorithm 3.1 is based on two loops in order to process all steps of all templates. In practice, all steps of all templates are processed only one time. Thus, the running time of Algorithm 3.1 using the asymptotic notation is $O(|\text{STEPS}|)$, where $|$ STEPS $|$ represents the sum of all steps of all templates to be processed.

After the generation of the ALTS models from use case templates, the model of the main application and the model of the interruption must be connected. Algorithm 3.2 is responsible for connecting them. Basically:

- The procedure uses a Depth-First Search (DFS) strategy for traversing all locations of the main application, that is, the application to be interrupted;

- As the proposed strategy only considers interruptions to be possible after expected results, the only possible points of interruptions are exactly after the transitions with the *expectedResults* annotation;

- When a transition with an *expectedResults* annotation is found in the main application model, a new transition (with the *beginInterruption_X* annotation) is added to the first location of the interruption model. And, from each final location of the interruption model, a new transition (with the *endInterruption_X* annotation) is added for connecting this model with the main application model;

- *Search* is a procedure responsible for marking a location (its second parameter) as a visited location and putting all adjacent locations in a list (its first parameter) to be processed.

Algorithm 3.1: Procedure that Translates Use Case Templates to an ALTS

```
UseModel UseCase2ALTS(Collection Templates) {
  UseModel alts := new UseModel();
  for each template in Templates {
    if (template.getFromStep() $\neq$ START) {
      // location after expected results of from step
      currentLocation := alts.getFinalLocation(template.getFromStep());
    } else {
      // associates a location to the step that creates it
      currentLocation := new Location(template.getFirstStep());
    }
    for each step in template {
      Transition steps := step.getUserAction();
      Transition conditions := step.getSystemState();
      Transition expectedResults := step.getSystemResponse();
      if (steps in alts) {
        currentLocation := alts.getLocationAfter(steps);
      } else if (step = template.getFirstStep() AND
          template.getFromStep() $\neq$ START) {
        // Avoid duplicating a steps trans. from the same location
        currentLocation := alts.addFromStep(currentLocation, steps,
            template.getFromStep());
      } else {
        // Create a new location for adding the new transition
        currentLocation := alts.add(currentLocation, steps);
      }
      if (conditions $\neq$ emptyCondition) {
        currentLocation := alts.add(currentLocation, conditions);
      }
      if (step = template.getLastStep() AND template.getToStep() $\neq$ END) {
        // The target loc. for the system resp. trans. is already created
        currentLocation := alts.addToStep(currentLocation,
            expectedResults, template.getToStep());
      } else {
        currentLocation := alts.add(currentLocation, expectedResults);
      }
    }
  }
  return alts;
}
```

Algorithm 3.2: Procedure that Combines the Main Application Model with an Interruption Model

```
CombineALTSModels(UseModel mainApplication, UseModel intModel, Integer
    intCode) {
  List list := $\emptyset$; // List of transitions to be visited
  Collection finalLocations := intModel.getFinalLocations();
  search(list, mainApplication.getRootLocation());
  while (list $\neq \emptyset$) {
    transition := list.remove();
    targetLocation := transition.getToLocation();
    if (transition.isExpectedResultsTransition()) {
      targetLocation.addBeginInterruptionTransition(
          intModel.getRootLocation(), intCode);
      for each location in finalLocations {
        location.addEndInterruptionTransition(targetLocation, intCode);
      }
      intCode++;
    }
    if (targetLocation is not visited)
      search(list, edge.getToLocation());
    }
  }
}
```

The running time of Algorithm 3.2 using the asymptotic notation is $O(\mid Q \mid \cdot \mid F \mid)$, where $|Q|$ is the number of locations of the main application and $|F|$ is the number of final locations of the interruption model.

3.3.2 Interruption Test Case Generation

This subsection describes the interruption test case algorithm developed to extract test cases from ALTS models. A test case generated from an ALTS is defined as follows.

Definition 3.4 (Test Case). *A test case is an ALTS $TC = \langle Q^{TC}, A^{TC}, L^{TC}, q_0^{TC}, T^{TC} \rangle$. The set of annotations is the same as the specification ($A^{TC} = A^S$) and the set of labels is $L^{TC} = L_I^{TC} \cup L_O^{TC} \cup L_C^{TC}$, where $L_I^{TC} \subseteq L_O^{SUT}$ (outputs of the SUT are the inputs of*

Algorithm 3.3: Test Case Generation Algorithm

```
Decompose(Location loc, Path path, Integer intCode) {
  if (loc.isLeaf() OR (loc.isRoot() AND path $\neq \emptyset$)) {
    // End of a path
    recordTestCase(path);
    return;
  }
  for each descendent in loc.getAdjacencies() {
    edge := getEdgeBetween(loc, descendent)
    if (edge.isBeginInterruption()) {
      intCode := edge.getIntCode();
    }
    if ((edge.getIntCode() = $-1$ AND edge $\notin$ path) OR (intCode >= 0 AND
        edge.getIntCode() = intCode)) {
      path.add(edge);
      if (edge.isEndInterruption()) {
        intCode := $-1$;
      }
      Decompose(descendent, path, intCode);
    } else if (edge.getIntCode() = intCode) {
      recordTestCase(path);
    }
  }
  return;
}
```

the TC), $L_O^{TC} \subseteq L_I^{SUT}$ (TC emits only inputs allowed by the SUT), and $L_C^{TC} \subseteq L_C^{S}$ (the conditions are the same specified by the specification). ◇

Test cases can be obtained from ALTS models, using the DFS method, by traversing the ALTS starting from the initial location (see Algorithm 3.3). As a general coverage criterion, all transitions need to be covered, i.e., all transitions of the ALTS model are visited at least once. As Algorithm 3.3 is based on DFS, its running time using the asymptotic notation is $O(|\,Q\,| + |\,T\,|)$, where $|\,Q\,|$ is the number of locations and $|\,T\,|$ is the number of transitions of the ALTS model.

Algorithm 3.3 requires three parameters: *loc*, a location of the model, indicating the current one during execution; *path*, a set of transitions from the model, indicating the path visited during the processing; and *intCode*, the interruption code, indicating that a given interruption is being processed.

The extraction is started from the root (the initial location of the ALTS model), verifying if the current location indicates the end of a path in the model, indicating that the test case has been extracted. In this case, it needs to be recorded. If the current location does not indicate the end of a path, then each of its descendants is visited through the depth-first search strategy.

To visit each of its descendants, the edge between the current location and its descendant is analysed. The search proceeds only if (Algorithm 3.3, Line 12): (i) the edge does not belong to the current analysed path, i.e., the edge has not already been “visited"(note that when the algorithm is processing the main application, the value of *intCode* is -1); or (ii) if it is an edge from an interruption behaviour (an edge with the *endInterruption_X* label). This precaution is necessary because after the interruption, the extraction process in the ALTS comes back to previous location (the last location of the main application before the interruption), therefore being possible to pass through the same interruption, in different parts of the model, and constraining that would cause inconsistency.

Due to these conditions, two scenarios are encountered: (1) Conditions (i) and (ii) are not satisfied: The search stops, recording the entire path as a test case avoiding loops in the main application and finishing an interruption with the correct *endInterruption_X* transition. In this case, the recursion step of the algorithm returns to the next branch that needs to be analysed, continuing the algorithm; (2) Condition (i) or (ii) is satisfied: The edge between the location and its descendent is added to the test case and the algorithm continues until it finds the end of the path, which happens when either a leaf in the graph or an edge going back to the root of the model are found.

These constraints over the extraction, when using the depth-first search approach, are required to avoid an explosion of paths during the test case extraction caused by loops in the ALTS model. This may reduce the number of extracted test cases, but without those constraints, the number of paths extracted becomes unfeasible, while most of them may be obtained by combining the extracted test cases. Also, from a functional testing point of view, in practice these excluded paths generally add redundancy to the test suite, that is, they do not generally add test cases that would uncover escaped faults because traversing the same loop several times to generate tests produces test suites with similar test cases [30]. Fully exploring loops it is usually a goal of other testing stages such as stress testing which is out of the scope of this work. Furthermore, by considering them, the algorithm would produce a large, infinite and not practical suite.

3.3.3 Interruption Test Case Selection

Exhaustive interruption test case generation is impractical due to the large amount of generated test cases. Particularly, in the mobile phone applications context, the majority of test cases are manually executed. In this scenario, test case selection strategies are much needed. The strategy used to reduce the test suite is a test case selection based on purposes. This strategy focuses on a coverage selection criterion, the test purpose, in order to test a particular system functionality [41; 50; 51; 83; 94]. The defined test purpose is used to filter out the model, that is, it is used to remove all paths that do not lead to the desired behaviour to be tested. After that, the generation algorithm is executed, for then, generate the test cases. Formally, a test purpose can be defined as follows.

Test purposes can be specified using a simple notation, where they are defined through transition sequences. In these sequences, an “*” (asterisk) indicates that, at this point, any transition can occur. A test purpose always finishes with a transition that has an *Accept* label (indicating that all test cases need to be in conformance with the purpose) or a *Refuse* label (otherwise).

Definition 3.5 (Test Purpose). *A test purpose is a deterministic LTS* $TP = (Q^{TP}, L^{TP}, q_0^{TP}, T^{TP})$*, equipped with the special labels Accept, Refuse, and “*”, and with the same alphabet as the specification, i.e.,* $L^{TP} = L^S$*.* Q^{TP} *is a countable, non-empty set of states,* $q_0^{TP} \in Q^{TP}$ *is the initial state, and* T^{TP} *is the transition relation.* ◇

Some hints on how to define test purposes are presented below:

- Choose the behaviour to be observed in the implementation and identify its description in the specification;

- If the behaviour to be observed is the first behaviour of the specification, then the test purpose should start with the description of this behaviour. Otherwise, add an asterisk followed by the description of the behaviour to be observed. This indicates that any behaviour can occur before the observation of the desired behaviour;

- If there are more behaviours to be observed in the same test purpose, go back to the first step;

- If the last behaviour description added to the test purpose is the last behaviour of the specification, then go to the next step. Otherwise, an asterisk should be added to the test purpose. This indicates that any other behaviour can occur after the desired behaviour;

- The last step is to add an *Accept* or a *Reject* label to the test purpose. As mentioned before, the *Accept* label is used to indicate that all generated test cases must be in conformance with the test purpose. The *Reject* label is used otherwise.

As an example of a test purpose, we will use that ALTS model from Figure 3.7 in order to define a test purpose for a scenario where a message is not removed because it is blocked. For this scenario, the following purpose could be defined: “*;‘Blocked messages cannot be removed’ dialog is displayed;*;Accept”. The LTS model that represents this test purpose is showed in Figure 3.9.

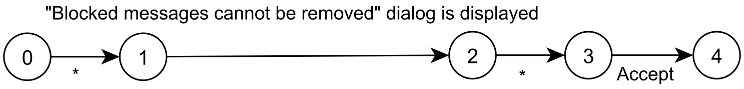


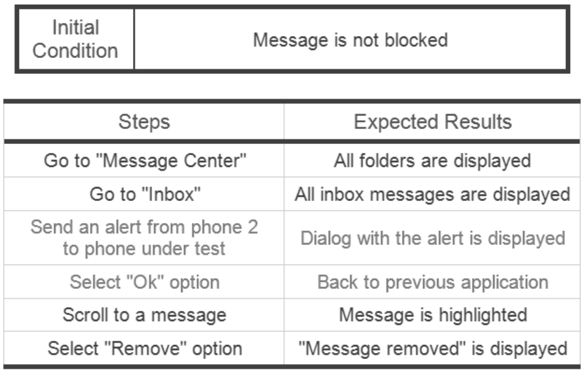
Figura 3.9: LTS Model of a Test Purpose

It is very simple to define test purposes where an interruption can occur. Given that the behaviour to be interrupted has been chosen, the name of the interruption must appear immediately after the description of this behaviour in the test purpose. The ALTS model with interruptions from Figure 3.8 will be used to demonstrate how to define test purposes to check specific interruptions. A test purpose will be defined to test the scenario where an alert appears when the user is accessing the inbox folder. This scenario can be specified through the following test purpose: “*;All inbox messages are displayed;Incoming Alert;*;Accept”.

Considering the defined test purpose, the model from Figure 3.8 is filtered out to be in accordance with it. So, the following edges of the model are removed: *beginInterruption_0*, *endInterruption_0*, *beginInterruption_2*, *endInterruption_2*, *beginInterruption_3*, *endInterruption_3*, *beginInterruption_4*, *endInterruption_4*, *beginInterruption_5*, and *endInterruption_5*. The last step is to execute the test case generation algorithm.

All presented algorithms are implemented in the LTS-BT tool [29]. In order to make the test execution activity easier, considering that this activity is manual, the tool generates test cases in an alternative representation instead of ALTS. Each selected test case is transformed in a matrix, where each condition is considered as an initial condition to execute the test case.

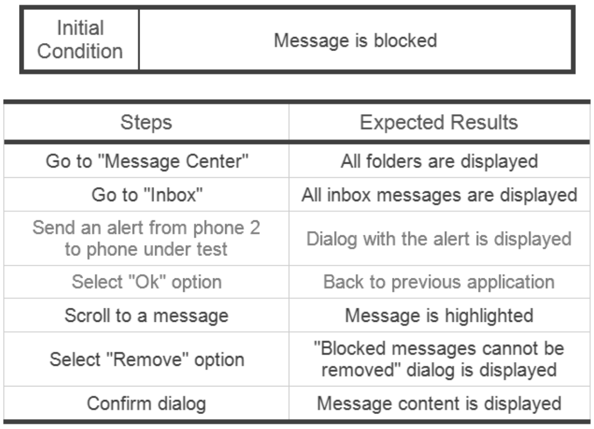
Figures 3.10 and 3.11 present the generated interruption test cases for the example above (the scenario where an alert appears when the user is accessing the inbox folder). Note that, in both generated test cases, the interruption occurs when the user is viewing the inbox folder, as was specified by the test purpose. Moreover, all scenarios of the main feature are covered. In the test case of Figure 3.10, an interruption occurs in the scenario where the message is removed, whereas, in the test case of Figure 3.11, an interruption occurs in the scenario where the message is not removed because it is blocked.



Initial Condition	Message is not blocked

Steps	Expected Results
Go to "Message Center"	All folders are displayed
Go to "Inbox"	All inbox messages are displayed
Send an alert from phone 2 to phone under test	Dialog with the alert is displayed
Select "Ok" option	Back to previous application
Scroll to a message	Message is highlighted
Select "Remove" option	"Message removed" is displayed

Figura 3.10: Test Case 01



Initial Condition	Message is blocked

Steps	Expected Results
Go to "Message Center"	All folders are displayed
Go to "Inbox"	All inbox messages are displayed
Send an alert from phone 2 to phone under test	Dialog with the alert is displayed
Select "Ok" option	Back to previous application
Scroll to a message	Message is highlighted
Select "Remove" option	"Blocked messages cannot be removed" dialog is displayed
Confirm dialog	Message content is displayed

Figura 3.11: Test Case 02

Notably LTS-BT allows for a systematic and less error-prone coverage of all possible interruptions automatically, since the tester does not need to specify all possible points where an interruption can occur – this is assumed by the tool. Moreover, LTS-BT makes it easier to focus on particular points to be interrupted and interruptions. These LTS-BT characteristics allow the tester to obtain test cases in a faster and reliable way.

It is important to remark that in the current version of the LTS-BT tool, test cases are selected as paths in the ALTS model. Therefore, even though the model is capable of representing non-determinism, the tool suits only deterministic applications, differently from the TGV tool [62], where a test case is a graph that can represent non-determinism. This is currently being addressed for the next versions of the tool.

As an example of a test case execution, the steps required for executing the test case of Figure 3.11 are: (1) the first step is to satisfy the initial condition, then an inbox message must be blocked for the test case execution. Moreover, as the test case needs an incoming alert interruption, two mobile phones must be available for the test case execution; (2) with the phone under test, the TESTER begins to execute the actions described in the test case. Thus, the “Message Center"application is started and, as result, all folders are displayed; (3) next, the second action is executed: the “Inbox"folder is selected and all inbox messages are displayed; (4) at this moment, the TESTER must cause an incoming alert interruption. So, he takes another phone and sends an alert to the phone under test. As expected result, a dialog must appear at the phone under test; (5) the TESTER selects the “Ok"option at the phone under test and control goes back to the previous application. In this case, all inbox messages must be displayed again; (6) the TESTER scrolls to a message and it is highlighted; (7) the “Remove"option is selected by the TESTER and the following message is expected: “Blocked messages cannot be removed"; (8) the TESTER confirms the dialog and the content of the message is displayed. Finally, if for all steps of the test case, the expected results were observed, the execution finishes with a pass verdict.

3.4 Properties of the Interruption Test Cases

This section comments on properties of the interruption test cases generated by the test case generation algorithm presented in the last section. Considering the execution of a test case against a SUT, three kinds of verdicts can be obtained indicating that the SUT should be approved or not: if the SUT emits the specified outputs for each input emitted by the test case, the verdict is *Pass*; if at least one of the outputs of the SUT is not specified by the specification, the verdict is *Fail*; and the *Inconclusive* verdict is emitted when the observed behaviour of SUT conforms to the specification but the behaviour described by the test purpose is not exhibited by the SUT.

It is very important to formalise the execution of the test cases in order to establish some properties as soundness and exhaustiveness, where the conformance relation is linked to verdicts obtained during the test execution [62]. Interruptions are clearly asynchronous events, but as we are considering a test architecture where the environment is fully controllable by the tester, all interruptions can be analysed as synchronous events. Thus, test cases interact with the SUT through a synchronous communication, where the execution of a test case against a SUT is modelled by a parallel composition with synchronisation on common actions. Basically, parallel composition is defined by the following rule:

$$P \mid\mid Q = \frac{p \overset{a}{\rightarrow}_P p',\ q \overset{a}{\rightarrow}_Q q'}{(p,q) \overset{a}{\rightarrow}_{P||Q} (p',q')}.$$

Considering the defined model of test case execution, each trace $\sigma \in$ Traces$(TC \mid\mid$ SUT) is associated with one of the following scenarios:

- If, at any moment, any unspecified output is emitted by the SUT, the execution is stopped and the resulting verdict is *Fail*, that is, verdict(σ) $=$ *Fail*;
- If the SUT, at any moment, blocks or spends a lot of time to emit an output, the resulting verdict is *Inconclusive* (a timer must be used in this case). So verdict(σ) $=$ *Inconclusive*;
- If the outputs of the SUT are specified by the specification but the behaviour specified by a test purpose is not exhibited, the resulting verdict is *Inconclusive*, that is, verdict(σ) $=$ *Inconclusive*;
- If all steps of the test case are executed and all expected results are observed, then the resulting verdict is *Pass*, i.e. verdict(σ) $=$ *Pass*.

Given the possible situations with their respective verdicts, the rejection of a SUT by a test case TC is defined as follows.

Definition 3.6 (may reject). *TC may reject SUT* $\overset{\Delta}{=} \exists\sigma \in$ *Traces*$(TC \mid\mid$ *SUT) : verdict(σ)* $=$ Fail. $\diamond$

The conformance relation of a SUT with respect to a specification S is decided based on verdicts obtained with the execution of the generated test cases. So, the next definition formally relates the previously defined conformance relation (Definition 3.3) to the verdicts of these executions considering some properties of test cases and test suites.

Definition 3.7 (Soundness and Exhaustiveness)**.** *A test case TC is sound for S and **conf** if $\forall$SUT, SUT **conf** $S \Rightarrow \neg(TC$ may reject SUT). A test suite is sound if all its test cases are sound and it is exhaustive for S and **conf** if $\forall$SUT, $\neg$(SUT **conf** $S) \Rightarrow \exists TC : TC$ may reject SUT. Finally, a test suite is complete if it is both sound and exhaustive.* ◇

Informally, a test suite is said to be sound if all correct implementations, and possibly some incorrect implementations, pass in the test (a sound test suite never rejects a correct implementation). On the other hand, a test suite is said to be exhaustive if all non-conforming implementations, and possibly some correct implementations, will not pass in the test. A test suite that can identify all conforming and non-conforming implementations is called complete.

A complete test suite is a very strong requirement for practical testing. Then, weaker requirements are accepted. In practice, sound test suites are more commonly accepted, since rejection of conforming implementations, by exhaustive test suites, may lead to unnecessary debugging. In this context, the test cases generated by LTS-BT have some properties stated in Theorem 3.1.

Theorem 3.1. *For every specification S, all test suites generated by the approach proposed in this chapter are sound. Moreover, the test suites can be considered as being exhaustive when they are generated using test purposes.* ◇

The proof of Theorem 3.1 is not detailed here but the main ideas are discussed (see detail proofs in Appendix A). For soundness, we need to prove that if a test case TC may reject a SUT (implementing the specification S), then $\neg$(SUT **conf** S). In this case, we only need to prove that a *Fail* verdict of a test case only occurs if the SUT emits an unspecified output. This was already discussed in this section and the unique case where a *Fail* verdict is obtained during a test case execution is exactly when the SUT emits an unspecified output. For exhaustiveness, we need to prove that for every non-conforming SUT there is a test purpose TP and a way of generating a test case TC from S and TP, such that TC may reject SUT. Given that ¬(SUT **conf** S), then there is a trace σ of S such that an output of SUT after σ is not allowed by S. So, the trace σ can be used to define a TP, after that, this test purpose can be used to generate test cases where the SUT may be rejected.

3.5 Case Study

The objective of this section is to present a case study performed in order to evaluate a practical application of the approach proposed in this work. As previously said, a scenario where interruptions are allowed may have infinite test cases. Thus, in practice, only a subset of interruption test cases are manually generated and executed. Considering this context, the main goal is to compare the manual process of test case generation with the automatic process based on the algorithms presented in Section 3.3 and implemented by LTS-BT [29]. Moreover, as the amount of test cases is large, some test case selection strategy is needed. Particularly, in this case study, the strategy used to select the test suite is based on test purposes defined in order to cover a fault model which describes the set of known defects found in the past [18]. Testers use fault models to define effective test cases since the test cases are specially defined to uncover defects that are likely to be present. The use of fault models for comparing testing approaches is important in our context because only stable versions of the software are available, that is, versions with all known defects already removed. In this case, fault models allow to compare testing approaches by observing if defined test cases would uncover defects or not.

This case study was performed using applications of the mobile phone domain whose descriptions are presented in the next subsection (Subsection 3.5.1). Subsection 3.5.2 describes how the case study was defined and conducted. Finally, Subsection 3.5.3 presents the results obtained during the case study execution.

3.5.1 Overview of the Case Study Applications

This subsection briefly describes the features used during the case study. All of them are reactive applications of the mobile phone domain. In summary, the description of the features is:

Tabela 3.1: Features

Features	Number of Use Cases	Number of Scenarios
Aircraft Mode	7	22
Incoming Call	1	2
Incoming Message	1	2
Alarm Clock	1	3

Aircraft Mode This is the main feature of the case study and it is the feature that must be interrupted. Aircraft Mode feature provides the functionality of allowing the user to turn off the radio frequency transceiver and still be able to use the applications of the phone. This feature allows the user to use applications of the phone while flying in an aircraft, but without receiving calls, messages, and so on.

Incoming Call This is an interruption feature. Incoming Call feature provides the functionality of receiving calls.

Incoming Message This is an interruption feature. Incoming Message feature provides the functionality of receiving messages.

Alarm Clock This is another interruption feature. Alarm Clock provides the functionality of generating alarm notifications based on specific time chosen by the user.

Table 3.1 shows some metrics of the features in order to illustrate their complexity such as the number of use cases and the number of possible scenarios. It is important to remark that, considering the relationship between all features of this simple case study, the amount of interruption test cases is more than forty million tests.

3.5.2 Case Study Definition

In this subsection, we present the evaluation criteria and fault model defined for conducting and evaluating the case study. The focus is on interruption testing and generation of test suites for manual execution.

The main goal is to show evidence on the benefits of automation in the interruption test case generation and selection process using the implemented algorithms. The strategy adopted to achieve the goal is to compare the manual process of test case generation with the automatic process proposed. In practice, this kind of testing is often conducted by manual processes of selection guided by expertise. Also, there are no related proposals of more systematic strategies that could make a good basis for comparison.

As the amount of interruption test cases is very large, generation and selection is guided by a fault model specification that indicates the kind of faults that can be usually be found in this kind of applications. Thus, test cases must be selected, both in manual and in automatic, with the objective to cover the whole fault model specification. The main metrics to be observed are: (1) the time spent during the test case generation and selection and (2) the coverage of an instance of the fault model specification that contains actual faults detected in products composed of these features.

The case study was conducted by three testers: one based on automatic process (named Tester 1) and two based on manual process (named Tester 2 and Tester 3). Considering the knowledge of the testers, they had good test skills and none of them knew the features under test before the case study execution. Thus, all testers performed the case study based on two kind of information:

- The specification of the features under test;
- The fault model profile.

The specification of the features under test is a document describing all use cases (Table 3.1) according to that notation presented in Section 3.1. It is important to mention that this document was prepared in five hours. On the other hand, the fault model profile was defined based on common problems related to feature interruptions and actual defects related to the features under test. The fault model given to the testers is specified in natural language and its description is defined as follows:

- After an interruption, the interrupted application does not maintain data entered by the user;
- After an interruption, the interrupted application does not continue its execution of the same point where it was interrupted;

Tabela 3.2: Metrics

Metrics	Tester 1	Tester 2	Tester 3
Preparation time	2 h	2 h	2 h
Generation time	80 *min*	165 *min*	150 *min*
Number of TCs	115	15	12
Productivity	86,5 TCs/h	5,4 TCs/h	4,8 TCs/h
Fault model coverage	57,14%	28,57%	28,57%
Number of invalid TCs	0 (0%)	4 (26,66%)	4 (33,33%)
Number of ineffective TCs	23 (20%)	13 (86,66%)	7 (58,33%)
Smallest TC (number of steps)	3	5	6
Biggest TC (number of steps)	12	11	8
Most common TC size	5	6 and 8	8

- Possible conflicts related to the use of shared resources (screen, network, and so on);
- Problems related to interruptions immediately before enabling the aircraft mode;
- Problems related to interruptions immediately after disabling aircraft mode;
- Problems related to interruptions when the aircraft mode is enabled.

By measuring coverage of an instance of this specification with actual faults (instead of coverage of the specification), where one kind of fault may correspond to more than one actual fault, it is possible to analyse which approach can be more effective to systematically investigate the implementation by generating a more complete test suite.

3.5.3 Case Study Results

This subsection presents and discusses the obtained results. Table 3.2 presents the metrics collected during the case study execution.

The first step of the case study consisted in reading the specification and the fault model. These documents are usually constructed prior to the testing process by requirements and quality engineers. In this sense, all testers had the same preparation time (Table 3.2, line

"Preparation time"), and as previously said, Tester 1 generated the test suite through an automatic process by using LTS-BT [29], and Tester 2 and Tester 3 generated the tests through a manual process. According to the results, from Table 3.2, Tester 1 generated the tests in less time (line "Generation time"). The generation time of Tester 1 basically consisted of the time needed to define test purposes once each LTS-BT execution consumed less than one second to generate test cases. Considering the generation time of Tester 2 and Tester 3, they spent more time because all test cases were manually selected and written in the format shown in Figures 3.10 and 3.11. Furthermore, Tester 1 generated the largest test suite (line "Number of TCs", where TCs means Test Cases) implying in more productivity (line "Productivity"). The productivity was calculated by observing the number of test cases generated per hour.

Considering the fault model coverage (Table 3.2, line "Fault model coverage"), Tester 1 reached the best coverage. It is important to remark that the percentage of fault model coverage achieved by Tester 2 and Tester 3 are equal but the faults found by them are not the same. Nevertheless, the set of faults found by Tester 1 contains all faults found by the other testers. This result is expected as LTS-BT allows a more systematic test case generation process, from the same base specification. However, as the process is guided by test purposes defined by the tester, the fault model coverage depends also on the tester's experience.

On the other hand, test cases generated through manual process are error-prone. The line "Number of invalid TCs" of Table 3.2 shows the number of test cases generated with errors, that is, test cases impossible to run, mainly because they miss information. Moreover, manually generated test suites tend not to take all scenarios of an interruption into account. This does not occur in the automatic process because when the tester decides, for example, to check the incoming call interruption at some point of the feature under test, the developed algorithms consider all scenarios of the interruption, for example, when the call is accepted and when it is rejected by the user.

Considering the number of test cases that actually do not find defects (line "Number of ineffective TCs"), Tester 1 reached the best results. The number of ineffective test cases also considers the invalid test cases. Finally, the three last lines of Table 3.2 give information about the size of generated test cases. Note that test cases generated by both strategies are similar w.r.t. size, that is, number of steps. This is explained by the fact that test cases were generated based on a structured document that may induce the same general kind of test cases to be defined. In practice, manual testing is not usually based on structured documents and then test cases tend to be as simple as possible. However, not using the same input document would put a threat to validity of the results, in the sense that with the same inputs, both strategies had the same basic information available.

In the scope of this case study, it is possible to conclude that the proposed strategy and tool allow a more systematic test case generation process contributing to better productivity and effectiveness of test process, depending on the tester's experience. Moreover, some problems of the manual process such as erroneous test cases generation is solved by the automatic process since all generated tests are sound (Section 3.4). However, note that Tester 1 did not reach 100% coverage. For this, it is necessary to define complete test purposes regarding the fault model. As this depends on the tester's experience and accuracy, it is possible that he can miss behaviour that should have been considered.

It is important to remark that the problem of selection in the scope of integration testing in general is a hard one. The possible number of test cases resulting from different combinations is large and also the set of all combinations is usually intractable by manual investigation. As a consequence, the achieved level of fault coverage depends greatly on information and expertise available to pinpoint the key test cases, for instance, common faults detected in a domain. For instance, only 57% of faults were covered by the test suite generated. However, note that this further exceeds the manually generated suites. Even though the testing strategy presented in this chapter is based on automatic generation, it also allows the experienced tester to target the selection process by defining the test purposes in a systematic way.

3.6 Related Work

This section presents some works related to our proposal. Lorentsen et al. [88] propose a way of identifying categories of interactions and create behavioural models that capture those interactions, where interruptions are a type of interaction. They use Coloured Petri Nets to manually model the interactions and a model checker for interactive graphical simulation. As disadvantages, the process is manual and the work is not devoted to testing.

Another interesting work is that belonging to Jard and Jéron [62], where the TGV tool is presented. TGV receives a specification and a test purpose as input and produces abstract test cases as output. The TGV input format for both specification and test purpose is IOLTS (already defined in Subsection 3.2.1). As mentioned in Subsection 3.2.1, it is possible to represent interruptions through IOLTS models. So the TGV tool can be used to generate interruption test cases, but an interruption behaviour needs to be replicated if it can occur at more than one place. Moreover, it is not possible to directly represent conditions associated to actions and due to the fact that the same interruption behaviour is replicated in the IOLTS model, the test purpose must specify the point where we want to verify the interruption and all other points where the interruption cannot occur. Thus, given that the tester needs to manipulate LTS models in the definition of the TGV test purposes, this notation is not useful in practice for interruption testing.

One possible solution is to consider the tool set proposed by the AGEDIS project that can generate test cases from high level models (e.g. UML diagrams) [57], where TGV is internally used to generate test cases. However, the tool set does not support interruption specifications directly as well as the newest version of UML (UML 2.0) with its greatly improved diagrams. In this case, the difficulties with interruption modelling and test purposes definition remain.

The process algebra CSP (Communicating Sequential Processes) was designed for describing systems of interacting components [109]. CSP has a specific operator for describing interruptions but its semantics is very different from the high (application) level interruption notion addressed in this chapter. The CSP interruption operator specifies that when a process P_1 is interrupted by another process P_2, the process P_1 is discarded and P_2 begins its execution. In our context, the process P_1 executes again after the execution of the process P_2. Jovanovic et al. [68] have proposed an extension of CSP to represent this kind of behaviour but there is not any tool supporting their proposal. Figueiredo et al. [39] present a behavioural model that represents interruptions in CSP without using the interruption operator, but the presented model is more suitable for representing the semantics of interruption test behaviour as presented in this chapter. Nogueira et al. [102] propose an approach to test case generation based on CSP. The SUT is specified using the same use case template presented in Section 3.1 and it is automatically translated to CSP using the strategy presented in [26]. The main objective of the work presented in [102] is to provide a strategy for testing individual features and feature interactions. Moreover, test purposes are defined in low level using CSP. The interruption testing is not directly treated, but the work can be adapted to test interruptions considering that all the possible points of interactions are specified in advance. In our approach, all points of interruption do not need to be explicitly specified, making the work of the tester easier and less error-prone.

Furthermore, the work presented in [25; 34] propose a strategy to reduce the test suite size based on test cases prioritisation. This requires that every individual pair of interactions is included at least once in a test suite. In this case, if it is not possible to execute the entire test suite, the tester can execute at least the most important test cases. But note that the prioritisation information is given by the tester. The strategy presented in this chapter is similar, in the sense that the focus is on particular interruptions by using a test purpose, but due to the expressiveness of test purposes, it is also possible to focus on and/or exclude particular functionalities associated with the interruptions.

3.7 Concluding Remarks

This chapter presented an approach to interruption testing that is based on a model capable of representing interruptions for reactive systems. The model makes it possible for interruptions to be combined at different points of possibly different flows of execution. This model is supported by the LTS-BT tool along with a test case generation algorithm and a test purpose-based selection technique. Test selection is crucial for interruption testing since the number of possible test cases is enormous. Also, in practice, not all possible points of interruption are fault-prone.

It is important to mention that the current version of LTS-BT is restricted to deterministic systems. This may seem unrealistic. However, particularly, if embedded systems such as mobile phone applications are considered, the tool can be largely applied. For these systems, applications are often deterministic ones that run on single-processor, single and restricted screen, and so on. However, they have complex patterns of interruptions which clearly justify the need for modelling and systematic test selection. Furthermore, ALTS models are capable of representing non-determinism and the algorithms can be clearly extended to support non-deterministic systems since the semantics of ALTS and IOLTS are very similar.

Capítulo 4

Trabalhos Relacionados e Problemas Identificados

Este capítulo apresenta uma revisão dos trabalhos relevantes no contexto de teste de sistemas de tempo real voltados para a geração de casos de teste a partir de especificações onde variáveis e parâmetros são permitidos. Como a pesquisa voltada para o teste de sistemas de tempo é muito recente, há poucos trabalhos relacionados com esse desafio. Após a análise dos trabalhos relacionados, várias limitações e problemas em aberto são identificados.

4.1 Related Work

Since strategies to generate real-time symbolic test cases are practically nonexistent, the focus here is to describe approaches related to testing real-time systems (not exactly symbolic testing of real-time systems) and argue why they are not considered as symbolic testing strategies.

4.1.1 Cardell-Oliver

Cardell-Oliver's work [28] addresses the problem of conformance testing for real-time systems and proposes an approach, based on UPPAAL timed automata specifications [82], to testing the same kind of system. As the UPPAAL timed automata have a dense or analogue-time model, Cardell-Oliver argues that their traces include behaviour which cannot be observed in an experiment. An example of an analogue-clock test case is: provide an input at time 1 and expect for the result at time 3. The tester implementing this test must be able to emit the input precisely at time 1 and check whether the output occurred precisely at time 3. In practice, the tester has finite-precision clocks and sample the outputs of the system under test periodically, e.g. every 0.3 time units, thus, it cannot distinguish between the output arriving anywhere in the interval (2.9, 3.1). In this sense, it is very difficult, if not impossible, to implement analogue-clock tests using finite-precision clocks. In this case, Cardell-Oliver proposes that a more appropriate model for observing real-time systems is a digital clock approximation.

As a TLTS representation of a TA can possibly have infinite states because of the representation of time, each timed trace of the TLTS is mapped into a set of possible integer-timed trace interpretations. Thus, symbolic states are used to represent a set of clock valuations. The symbolism is only used to abstract time and does not take the data of the system under test into account, so this proposal cannot be classified as a symbolic testing approach.

Furthermore, the paper considers a kind of test purpose, named test views, where the tester can select relevant events to observe. An implementation relation is defined based on trace equivalence under the assumption that the implementation is input-enabled (input-complete) and that it has no more states than the specification. An algorithm implementing the approach is presented, but, it seems that there is not a tool to support the work.

4.1.2 En-Nouaary et al.

En-Nouaary et al. [46] address the issue of testing real-time systems specified as a variant of the TAIO presented in Subsection 2.2.3. The TAIO considered in [46] is assumed to have instantaneous transitions, that is, once the transitions are enabled they must be taken immediately. This assumption reduces too much the expressiveness of the model, for example a simple specification such as “when an input is provided, an output must be generated within at most 10 time units” cannot be expressed.

En-Nouaary et al. propose a test case generation approach that is divided into three steps: firstly, as the semantics of a TAIO can be defined in terms of an infinite TIOLTS (see Subsection 2.2.3), the authors represent this infinite TIOLTS using a finite region graph where the locations symbolically represent a set of clock valuations. Secondly, each clock region of the region graph is sampled, according to a granularity, in a way that each clock region is transformed into a finite set of clock valuations; thus, the region graph is reduced to another graph, named grid automaton, which is then transformed into a timed finite state machine. Finally, they use state characterization techniques for test case generation [47].

Note that, as the work only represents time symbolically, it is not considered as a symbolic testing strategy, since the data of the system must be also symbolically taken into account. The work supposes that the implementation under test has the same number of locations as the specification. Thus, the implementation relation is based on trace equivalence. The adopted assumption is very strong compromising the usefulness of the work in practice. Moreover, no algorithms for test case generation are shown.

An interesting contribution of En-Nouaary et al. is the study about fault models. They argue that two types of faults are possible: timing faults and action and transfer faults [48]. Timing faults are related to violation of transition time constraints and action and transfer faults are similar to the classical faults of finite state machines. In [46], the authors discuss how the fault model can be used to test timed systems based on TAIO model.

In [44; 45], the testing approach is improved changing the last step of the test case generation process. Instead of transforming the generated grid automata into a timed finite state machine, the grid automata is traversed using an adaptation of the Depth-First Search strategy in order to generate test cases. In this more recent work, a test selection strategy based on test purposes is defined and algorithms are presented. Nevertheless, this improved approach only represents time symbolically.

4.1.3 Li et al.

Li et al. [87] propose an approach to property-oriented real-time test case generation. As specification language, they use time-enriched statecharts and provide a restricted real-time logic as the property specification language. Li et al. argue that statecharts cannot be easily manipulated for test generation, then they provide a way of transforming statecharts into extended finite state machines, from which test sequences are obtained.

Li et al. [87] focus only on specification languages. Thus, several concepts of a complete testing strategy are not taken into account such as assumptions related to specifications and implementations, conformance relation, test architecture, tools, case studies, and so on.

4.1.4 Khoumsi

Khoumsi [69; 70] proposes an approach to symbolic test case generation for real-time systems. To the best of our knowledge, this is one of the few approaches that tries to symbolically deal with time and variables and actions with parameters representing the system data. The Khoumsi's main objective is to combine a real-time testing strategy with a non-real-time symbolic testing strategy.

Khoumsi's work is based on that symbolic model theory presented in Subsection 2.1.9 extended with time. Basically, the proposed approach is divided into two steps: firstly, the real-time symbolic model is transformed into an automaton where the setting and expiration of clocks are represented as actions; finally, the symbolic testing approach presented in Subsection 2.1.9 is adapted to generate test cases.

In this approach, a new real-time symbolic model is proposed, named Timed Input-Output Symbolic Automata (TIOSA), but its semantics is not formally defined. Additionally, the first step of the approach, described above, restricts too much the use of clocks, guards, and clock resets leading to a less expressive and flexible specification language. Under the assumption of input-completeness, the adopted conformance relation can be considered as timed trace inclusion. Finally, there is no tool supporting the work and real case studies were not performed to validate the applicability of the proposed strategy.

4.1.5 Briones and Brinksma

Briones and Brinksma [23] present en extension of Tretmans' theory and algorithm [117] for testing real-time systems. A distinguishing characteristic of this work is that it takes quiescence into account and provides an operational interpretation of this concept in the context of RTS. Only output quiescence is considered. They consider an output quiescent state as one where the system is unable to generate an output without further input stimuli. Briones and Brinksma argue that the quiescence can only be detected by waiting for outputs, but as we cannot wait forever a maximal duration M must be defined. So, the work proposes a parameterised conformance relation where output quiescence only can be observed after a minimal delay of M time units. The work is based on TIOLTS, which serves as semantics for TAIO (see Subsection 2.2.3). Additionally, the paper defines the concept of real-time test cases considering execution and verdicts, and presents in an abstract way an algorithm for generating them. Considering that the underlying continuous model of time is represented through an infinite TIOLTS, the proposed algorithm generates uncountable test cases.

In [24], Briones and Brinksma extend the framework proposed in [23] in the sense of allowing the implementation to be sometimes non-input-complete. This new paper presents an extension of TIOLTS where input and output sets are divided in channels and in each reachable state each input channel is either blocked or all inputs are accepted, i.e. the implementation can sometimes be non-input-enabled. The general maximal duration M, cited above, is relaxed and they allow different bounds for different sets of outputs. Moreover, the entire framework is updated to deal with these extensions, including the algorithm for test case generation. But, the algorithm has the same problem as in [23], that is, it also generates uncountable test cases.

4.1.6 Bohnenkamp and Belinfante

Bohnenkamp and Belinfante [20] present an extension of TorX [119] where timing constraints can be expressed in the specification. TorX is an on-the-fly testing tool that tests for the conformance relation proposed by Tretmans [117]. Bohnenkamp and Belinfante's work is influenced by the framework presented in Subsection 4.1.5, that is, the main objective of this work is to provide an extension of TorX to implement the main ideas of the Briones and Brinksma's work [23].

To avoid the generation of uncountable test cases, Bohnenkamp and Belinfante adopt a symbolic representation of TIOLTS, where each symbolic location represents the maximal set of clock valuations that satisfy a given clock constraint. This strategy is not considered as a symbolic testing strategy, as only time is abstracted. The tool assumes input-completeness of the IUT and that the tool must run on the same host as the IUT. The latter restriction reduces the usefulness of the tool in practice, since several systems cannot be tested such as embedded systems with limited resources (e.g. smart cards, mobile phones, music players, and so on).

4.1.7 Bodeveix et al.

Bodeveix et al. [19] propose a way of checking real-time dependability requirements by means of testing. They adopted a particular kind of timed automata, where determinism and explicit inputs and outputs are assumed. The main idea is to model dependability requirements as test purposes. The strategy is described in three steps: (1) a kind of synchronous product is performed between the specification and the requirement to be checked; (2) the states of the resulting model are symbolically represented abstracting only time; (3) a reachability algorithm is executed to generate only one test case capable of checking the dependability requirement.

The paper is very short and important concepts of a complete testing framework are not discussed such as assumptions related to specifications and implementations, conformance relation, test cases, verdicts, oracles, and so on. Additionally, algorithms are not shown and tools were not developed to support the proposal.

4.1.8 Larsen et al.

Larsen et al. [80] propose a tool and the related theory for online testing of real-time systems. The work is based on non-deterministic timed automata with inputs and outputs (TAIO) specifications. However, the adopted TAIO is a variant of that presented in Subsection 2.2.3, where both locations and transitions can have guards (clock constraints). The developed tool was firstly named T-UPPAAL [100], but today its name is UPPAAL TRON (UPPAAL for Testing Real-time systems ONline). This tool was implemented by extending the UPPAAL model-checking tool [82].

A distinct characteristic of the work is the proposal of a formal implementation relation that takes environment assumptions into account. An environment assumption is a kind of test purpose of the property oriented testing theory, presented in Subsection 2.1.6. Thus, the main goal of the proposed implementation relation is to check if an implementation is in conformance with its specification when operating under some environment assumptions. According to Larsen et al. [80], modelling the assumptions separately has several advantages: (1) considering a specific environment, the testing tool generates only realistic test cases reducing the number of test cases and improving the quality of the test suite; (2) the testing process can be guided to specific scenarios of interest; (3) when the environment model is separated from the system model it is easier to test the system under different assumptions. The proposed implementation relation is based on Tretmans and de Vries' work [118; 40] and coincides with timed trace inclusion considering the input enabledness assumption.

As previously said, the work is based on a variation of TAIO, but Larsen et al. [80] show that the semantics of a TAIO can be defined in terms of an infinite TIOLTS. Thus, all the theories and algorithms presented are based on TIOLTS. In [80], the main algorithm for generating and executing test cases are presented as well as its proof of soundness and completeness. As a TIOLTS, representing a TAIO, can possibly have infinite states because of the representation of time, the generation of test cases uses a reachability algorithm that operates on symbolic states (a symbolic state represents a set of clock valuations). As only time is symbolically represented and the system data is not symbolically taken into account, we cannot consider this proposal as a symbolic testing approach.

The work reported in [80] presents an experiment to validate UPPAAL TRON and indicate the applicability of the proposed technique. As the test generation and execution is online, a drawback is the need to implement an adapter component to link the system under test to the UPPAAL TRON tool. In order to evaluate the algorithms and the tool in detail, a real-life application was tested and the results are described in [81].

4.1.9 Hessel et al.

Hessel et al. [60] present a strategy of time-optimal test case generation using the data structures and algorithms of the UPPAAL model checking tool [82]. Time-optimal test cases are defined as test cases guaranteed to take the least possible time to execute. According to Hessel et al. [60], time-optimal test cases are important for some reasons: (1) using time-optimal test cases, the total execution time of a test suite is reduced allowing more behaviour to be tested; (2) regression testing can be executed as quickly as possible; (3) the time-optimal test cases have high probability of detecting errors, considering that the fastest scenarios are stressful situations. The proposed strategy can generate test cases using manually defined test purposes or automatically generated from some coverage criteria.

The proposed strategy assumes that both the implementation under test (IUT) and the environment (the test purpose) are modelled using a deterministic and output urgent class of TA. A TA is deterministic when two transitions with the same label lead to the same state and it is output urgent if, when an output is enabled, it occurs immediately. These characteristics are very restrictive reducing the expressiveness of the model. The work uses the fastest diagnostic trace of the UPPAAL tool to generate time-optimal sequences. This functionality of UPPAAL generates a trace with the shortest accumulated time delay witnessing a submitted safety property. Then, a test case is generated from this diagnostic trace. As previously said in Subsection 4.1.8, the UPPAAL's reachability algorithm operates on symbolic states, but only time is symbolically represented. The implementation relation considered is the timed trace inclusion as in [80; 81]. One of the main ideas of the work is that test purposes can be formulated as safety properties that can be checked by the reachability analysis performed in a model generated by the combination of the specification with the test purpose. Another interesting idea is the use of coverage criteria to generate test cases. In this case, the proposed solution is to annotate the model using auxiliary variables to mark the coverage of target elements (e.g. edges and locations). The work also presents an interesting experiment, but it is clear that the annotation of the model with new auxiliary variables is tedious and error prone in practice. Moreover, the inclusion of these variables increases the state space reducing the applicability of the work in practice.

Several limitations of the work described above are solved in another work [61], which extends the requirement specification language of UPPAAL in order to allow the use of keywords to represent coverage criteria. Therewith, the reachability analysis algorithm of the UPPAAL tool was modified to eliminate the need of manually annotating the model and another version of UPPAAL was created, the UPPAAL CoVer tool. Even so, the defined language is still very restrictive and the specification model continues being deterministic and output urgent.

Finally, a book chapter [59] was written to describe both the online testing using UPPAAL proposed by Larsen et al. [80; 81] and the offline testing using UPPAAL proposed by Hessel et al. [60; 61].

4.1.10 Merayo et al.

Merayo et al. [96] present a formal framework to specify and test real-time systems that considers both hard and soft deadlines, where hard deadlines must be always met on time and soft deadlines can be sometimes met in different times. The model used to specify the software system is an extension of the classical finite state machine, called Timed Extended Finite State Machines (TEFSM). Transitions in classical finite state machines indicate that if a machine is in a state s and receives an input i then an output o will be produced and it will change its state to s' (this can be represented as $s \overset{i/o}{\rightarrow} s'$). The timed extension proposed is represented as $s \overset{i/o}{\longrightarrow}_{[t_1,t_2]} s'$ and it means that if a machine is in a state s and receives an input i then an output o will be produced and it will change its state to s' wasting a time greater than or equal to t_1 and smaller than or equal to t_2. Merayo et al. argues that, in the context of RTS, the notion of correctness has several possible definitions. For this, the framework defines several conformance relations. In practice, one may consider that a system under test is in conformance with a specification if all actions are performed exactly on a predefined time, while another could consider that the implementation has to be always/sometimes faster. Thus, depending on the situation, a different conformance relation can be taken into account. The presented conformance relations are based on Tretmans' work [117]. The paper also formalizes the concept of test cases and test suites and gives some directions of how to apply the test cases to an implementation. But, algorithms are not presented.

The work discussed above was extended in [97], which itself continues the proposal presented in [103]. In this more recent work, two kinds of time are considered: actions with associated time and time-outs. When time-outs are allowed the state of the system can change only with the time evolution (without the occurrence of actions). Moreover, an algorithm to generate test cases is presented and some small examples are discussed. In [98], Merayo et al. extends the work [97] to deal with stochastic time systems. Finally, the work presented in [99] extends all the developed formal framework to deal with specifications where time requirements are defined using intervals.

The ideas presented by Merayo et al. [96; 97; 98; 99; 103] are very interesting. It is extremely useful to have many conformance relations, so the theory can be used in different contexts. But, in general, the work has some disadvantages: (1) there is no a tool supporting the approach; (2) real examples and case studies are not presented; (3) the model does not allow to specify actions with parameters and communicating machines are not considered, thus it is not possible to deal with asynchronous events; (4) only one clock is used to control how time evolves reducing the expressiveness of the model; (5) only the discrete-time model is considered.

4.1.11 Krichen and Tripakis

Currently, one of the most complete works on testing of real-time systems is the one developed by Krichen and Tripakis [73; 74; 75; 76; 77; 78]. In [74], they propose a framework for conformance testing of real-time systems where specifications are modelled as non-deterministic and partially-observable timed automata. Krichen and Tripakis argue that, in practice, when the model is built compositionally, component interactions are usually non-observable by the tester and this abstraction often results in non-determinism.

In comparison with other approaches, Krichen and Tripakis' work uses less restricted timed automata. For example, several restrict assumptions are considered by other approaches such as isolated and urgent outputs (Subsection 4.1.9); the use of determinizable timed automata with restricted clock resets (Subsection 4.1.4); the use of trace equivalence as conformance relation, considering that the implementation has no more states than the specification (Subsections 4.1.1 and 4.1.2); permission of using only one clock in the specification (Subsection 4.1.10); and so on. Krichen and Tripakis use a kind of TAIO where each transition is annotated with one of the following three deadlines: lazy, delayable, and eager. The lazy deadline imposes no urgency, delayable means that once enabled the transition must be taken before it becomes disabled, and eager means the transition must be taken as soon as it becomes enabled.

They propose an extension of the conformance relation from [118], named timed input-output conformance (tioco). This new conformance relation is defined by including time delays in the set of observable outputs. They do not require the specification to be input-complete, thus tioco is more expressive than the other conformance relations such as trace equivalence (Subsections 4.1.1 and 4.1.2) and trace inclusion (Subsections 4.1.9 and 4.1.4). The proposed conformance relation is more expressive in the sense that it allows an implementation to accept inputs not accepted by the specification, whereas the other timed conformance relations above do not. Several characteristics of tioco are discussed in [77] such as transitivity, an extensive comparison with related conformance relations, the prove that checking tioco is undecidable (it is not a problem for black-box testing since the implementation model is unknown, the conformance cannot be directly checked) and that it does not distinguish specifications with the same set of observable traces.

Most of the other authors only consider analogue-clock tests, i.e. tests that are very difficult to execute in practice. Nevertheless, Krichen and Tripakis consider both analogue-clock and digital-clock tests. Analogue-clock tests can measure precisely the delay between two events, whereas digital-clock tests can only count how many time units of a periodic clock have occurred between two events [74]. They use symbolic reachability algorithms for test case generation, whereas other approaches use symbolic representation of time with classical reachability algorithms. There is a tool, named TTG, supporting the proposed strategy and in [74] only a toy example is used as case study.

The strategy of analogue and digital-clock test generation is improved in [75]. Most test generation algorithms rely on an implicit determinization of the specification during the test case generation, but this is a problem when analogue-clock tests are considered because timed automata are not determinizable in general [5]. In [74], Krichen and Tripakis proposed an on-the-fly determinization of the specification during the execution of the test, but the generated algorithm is costly and the tester must quickly respond to the outputs of the system under test. Thus, they proposed, in [75], a pragmatic approach where they suppose that the tester has a single clock and that it is reset every time the tester observes an action of the system. This proposed strategy allows analogue-clock tests to be represented as deterministic timed automata. The generation of digital-clock tests has a different problem: as this kind of test can be represented statically as finite trees, the generation can be offline or on-the-fly and the strategy adopted in [74] is to generate all possible tests up to a given depth, leading to an explosion of test cases. In [75], this problem is solved by providing a method to generate tests which cover the specification with respect to some criteria.

Krichen and Tripakis present, in [76], the proposed framework in a methodological point of view with emphasis on the expressiveness of the models and showing several examples of scenarios to be specified. A real example is used as case study in this more recent work. Finally, all the work discussed in this section is described in detail in Krichen's PhD thesis [73] and in [78].

4.1.12 Zheng et al.

Zheng et al. [124] address methods to generate test cases from formal specifications of real-time systems and provide a metric-based test selection method for sufficient testing of a given implementation. The systems are specified using an object-oriented approach: firstly, the abstract data types are separately defined; secondly, each reactive object of the system is specified using a kind of extended finite state machines; finally, the whole system is specified as a network of communicating objects.

As the specification notation allows time to be continuous, a grid is used to digitise the extended finite state machines. The grid is a covering of the underlying analogue time space, mapping points of that space onto a single representative of each grid region. So, this strategy also represents time symbolically as most of the other approaches discussed in this chapter. From grid automata, test cases can be generated according to a given coverage criterion (state or transition coverage) or according to a fault model.

The work also describes an experimental study where a test bed implementing the proposed strategy, named TROMLAB, is used to validate the work. But, the work has some drawbacks such as a conformance relation is not defined, algorithms and examples of test cases are not presented, the cited tool is not available, and the specification language only considers synchronous communication between objects.

4.1.13 David et al.

David et al. [37] propose a game-theoretic approach to the testing of real-time systems. Systems are modelled by Timed Input-Output Game Automata (TIOGA), which is a variant of timed automata (see Subsection 2.2.3) with their actions partitioned into controllable ones and uncontrollable ones. When an action is controllable it means that the tester determines when or which action will occur, whereas when an action is uncontrollable it means that it is the system under test that determines when or which action will occur. Considering that the set of actions are divided into input and output actions, David et al. assume all output actions to be uncontrollable and all input actions to be controllable. Test purposes can be defined as Timed CTL formulas.

According to David et al., the proposed strategy can be defined as follows: a play of the timed game between the system and the tester is a run of the TIOGA towards a specified test purpose; if the Timed CTL formula is satisfied by the TIOGA, it can synthesize a winning strategy; since a winning strategy is a guide towards the goal states, which satisfy the test purpose, it can be viewed as a test case. This strategy is implemented in a timed game solver, named UPPAAL TIGA.

David et al. show that the semantics of TIOGA can be defined in terms of TIOLTS and reuse the conformance relation defined by Krichen and Tripakis (Subsection 4.1.11). Internally, the strategy represents time in a symbolic way and uses model checking techniques to verify the satisfiability of a formula against a specification, and if so, a test case is generated. In this sense, this technique is not considered to be a symbolic testing strategy.

In [38], David et al. propose another game-theoretic approach to testing partially observable real-time systems. This work differs from the former because the tester may observe neither internal actions nor internal state changes due to these internal actions. Moreover, the tester has limited precision ways to analyse the SUT, which avoids knowing which state the SUT is in or the precise observation a timed trace. In this case, the SUT can only be observed through a finite number of possible observations. Finally, an observation-based conformance relation is proposed along with algorithms for test case generation, but only time is symbolically treated.

4.1.14 Adjir et al.

Adjir et al. [3; 4] propose a technique for conformance testing of real-time systems using TINA, a toolbox for the edition and analysis of Petri Nets and Time Petri Nets. According to the authors, the toolbox allows the generation of time-optimal test cases. As specification language, they use Prioritised Time Petri Nets. In this model, there is a priority relation on the transitions, that is, a transition can only be fired if it has the highest priority at the moment.

The authors mention that the proposed technique is based on timed trace inclusion, but no conformance relation is formally defined. Specific scenarios of testing can be selected through manually defined test purposes and covering criteria specified in the state-event linear temporal logic SE-LTL. Besides not presenting algorithms or case studies, this work only abstracts time.

4.1.15 Styp et al.

Styp et al. [120] propose a combination of symbolic transition systems introduced in [52] with timed automata presented in [5], named Symbolic Timed Automata (STA). The proposed formalism allows the modelling of real-time reactive systems with data input and output. It is possible to use variables as bounds in clock guards and associate clock invariants with locations. The semantics of STA is defined in terms of TLTS (see Subsection 2.2.3).

A new conformance relation is defined: stioco. The stioco relation is very similar to the tioco relation defined by Krichen and Tripakis (Subsection 4.1.11), but symbolic constraints (universally quantified formulas) are considered instead of dealing with concrete outputs. Thus, an implementation conforms to a specification for stioco, if, whenever the constraints are satisfied for the implementation to produce an output or delay, then also the specification satisfies the constraints to produce the same output or delay. The authors state that stioco coincides with tioco at the semantic level.

As the work is at the beginning, it is far from a complete testing approach. Test cases are not defined along with a test architecture to execute them. Test case generation and selection strategies are not presented. Moreover, there are neither algorithms nor tools to support the work proposed.

4.1.16 Timo et al.

Timo and Rollet [113; 114] propose a conformance testing approach to data-flow real-time systems based on a variant of timed automata in which only variable changing are considered as events.

In [114] the model called Variable Driven Timed Automata (VDTA) is proposed. In the states of a VDTA either the time elapses continuously or the environment modifies the values of input variables. All transitions are urgent and the values of output variables can only be observed. Moreover, a timed variable-change conformance relation (tvco) is proposed. Basically, an implementation is in conformance with its specification for tvco if all behaviours of the implementation are allowed by its specification. In this case, the implementation must change the value of input variables in a time allowed by the specification. An online testing algorithm is proposed, but test cases are not formally defined.

The approach presented in [114] is improved in [113]. Test purposes are introduced as a test case selection strategy and the time is treated with region graphs, which may lead to the state space explosion problem. Finally, abstract interpretation and approximation techniques are proposed to generate test cases, but no algorithms are presented. It is important to mention that only deterministic models are considered, quiescence is not discussed, and there is no tool supporting the work.

4.2 Comparison of Reviewed Work

This section concludes the analysis of related work with a comparison among all studied contributions. The analysis is divided into three tables because of the limited space. In the first table (Table 4.1), the following characteristics are considered: (1) if the strategy of test case generation is online or offline; (2) if the proposal allows test purposes specification; (3) if the proposed strategy is supported by tools; (4) the specification language used to model the IUT; (5) if the work takes quiescence into account.

Considering the first characteristic, most of the approaches adopt offline test generation. Few approaches that consider an online test case generation has a tool available. As discussed before, it is easier to deal with non-determinism through an online strategy. On the other hand, it is more difficult to guide the generation with test purposes. An interesting work is the one developed by Krichen and Tripakis (Subsection 4.1.11), where the generation is offline but the strategy takes non-determinism into account. The generated test cases can be seen as trees and the action of the tester depends on the observation history. It is important to remark that non-determinism is an important characteristic of the real-time context, since most of specifications are composed of parallel components.

The second characteristic is related to specification of properties to be verified during the test. Most of the approaches allow the specification of test purposes. In some cases, the approaches marked with an asterisk "*", the strategy allows the specification of the environment that interacts with the IUT emitting inputs and receiving outputs. In this case, the specification of the environment can be considered as a test purpose. Test purposes are extremely needed in the context of real-time systems because the available algorithms usually generate a huge amount of test cases.

As the researches in the context of real-time testing are very recent, there are few tools available. Basically, we can cite TROM (Subsection 4.1.8), CoVer (Subsection 4.1.9), TTG (Subsection 4.1.11), TIGA (Subsection 4.1.13), and TINA (Subsection 4.1.14) for effective testing of real-time systems. The other tools marked with an asterisk are only cited on their respective papers, but they are not available. The three tools TROM, CoVer, and TIGA are related in the sense that they are based on the UPPAAL model checking tool. These three tools use the UPPAAL notation as the input specification.

Considering the notation used as specification language, most of the approaches use models derived from timed automata [5]. In general, timed automata cannot be determinized [5], thus most approaches impose several restrictions to the specification language. Some authors completely disallow non-determinism such as Cardell-Oliver (Subsection 4.1.1), En-Nouaary et al. (Subsection 4.1.2), Li et al. (Subsection 4.1.3), Briones and Brinksma (Subsection 4.1.5), Bodeveix et al. (Subsection 4.1.7), Hessel et al. (Subsection 4.1.9), Zheng et al. (Subsection 4.1.12), and David et al. (Subsection 4.1.13); whereas, others restrict the use of clocks, guards, or clock resets such as Khoumsi (Subsection 4.1.4), Bohnenkamp and Belinfante (Subsection 4.1.6), and Merayo et al. (Subsection 4.1.10). The most expressive specification languages are used by Larsen et al. (Subsection 4.1.8) and Krichen and Tripakis (Subsection 4.1.11). Non-determinism is also important to model timing uncertainty, that is, it is more realistic to allow an output occurring in some interval of time. In this case, non-determinism is a choice between letting the time pass or emitting an output.

Khoumsi (Subsection 4.1.4) is one of the authors that proposes a specification language where parameters and variables containing data of the system can be defined. This is the first step in order to provide an effective real-time symbolic testing strategy. Nevertheless, strong restrictions are made on clocks, guards, and clock resets leading to restrictions on its applicability in practice. Two other approaches described in Subsections 4.1.15 and 4.1.16 are intended to provide a real-time symbolic testing strategy, but these approaches can be considered as incomplete since test cases are not formally defined, a test architecture is not defined, no algorithms are presented, and there is no tool supporting the work.

The last characteristic considered in Table 4.1 is quiescence. Quiescence is a characteristic of systems that indicates the absence of outputs, and as described in Subsection 2.1.5 it is extremely related to real-time systems. To provide an effective way of dealing with quiescence, the following concepts must take it into account: input specification, conformance relation, oracle, and so on. In this sense, only two approaches consider quiescence: Briones and Brinksma (Subsection 4.1.5) and Bohnenkamp and Belinfante (Subsection 4.1.6). However, the former work does not have an implemented tool, whereas the latter implemented a prototype which is unavailable.

Work	**Test Case Generation**	**TP**	**Tool**	**Spec. Language**	**Quiesc.**
Cardell-Oliver	offline	yes*	Essex*	TIOLTS	no
En-Nouaary et al.	offline	yes	no	deterministic and output urgent TAIO	no
Li et al.	offline	yes	no	RT Statecharts	no
Khoumsi	offline	yes	no	non-deterministic TIOSA	no
Briones and Brinksma	offline	no	no	TIOLTS	yes
Bohnenkamp and Belinfante	online	yes	yes*	non-deterministic safety TAIO	yes
Bodeveix et al.	offline	yes	no	a kind of TAIO	no
Larsen et al.	online	yes	TRON	TAIO (with guards on locations and transitions)	no
Hessel et al.	offline	yes	CoVer	deterministic and output urgent TAIO	no
Merayo et al.	offline	no	no	non-deterministic TEFSM	no
Krichen and Tripakis	offline and online	yes	TTG*	partially-observable and non-deterministic TAIO	no
Zheng et al.	offline	yes*	TROMLAB*	TEFSM	no
David et al.	offline	yes	TIGA	TIOGA	no
Adjir et al.	offline	yes	TINA	Prioritized Time Petri Nets	no
Styp et al.	no	no	no	STA	no
Timo et al.	offline	yes	no	VDTA	no

Tabela 4.1: Related Work

In the second table (Table 4.2), the following characteristics are considered: (1) definition of a conformance relation; (2) assumptions related to the specification; (3) assumptions related to the implementation under test. Considering the first characteristic, most of the approaches define a conformance relation based on either trace equivalence or trace inclusion. These kinds of conformance are very restricted because the implementation must have inputs and outputs defined in the specification. In the case where the specification does not completely specify a system, an implementation is allowed to have inputs not defined in the specification, thus only the outputs related to inputs defined in the specification must be considered. In this sense, Krichen and Tripakis (Subsection 4.1.11) propose a less restricted conformance relation, named tioco. Briones and Brinksma (Subsection 4.1.5) is the only work that defines a conformance relation considering quiescence and the idea is implemented by Bohnenkamp and Belinfante (Subsection 4.1.6). Merayo et al. (Subsection 4.1.10) define several conformance relations which can be considered as timed trace inclusions as well as the conformance relation defined by Timo et al. (Subsection 4.1.16). An interesting conformance relation is defined by Styp et al. (Subsection 4.1.15) based on symbolic constraints instead of concrete outputs, however at the semantic level it coincides with tioco.

Considering the second characteristic in Table 4.2, as already discussed, several approaches impose determinism to the specification in order to simplify the strategy. Almost all approaches assume input-completeness of the specification, so a complete specification of the IUT must be available. This restriction is relaxed by Krichen and Tripakis (Subsection 4.1.11) where the testing process can be performed with a system partially specified.

The last characteristic in Table 4.2 is related to the assumptions about the IUT. Practically, all approaches assume the input-completeness of the IUT. In practice, this assumption is true in many contexts, but not all. There are several scenarios where an IUT may not be input-complete, for instance, when a user tries to save a read-only text or to insert a PIN card in a slot of a cash machine where there is already another PIN card inserted. It is clear that there are situations where the inputs can be forbidden or ignored by the system. In this sense, Briones and Brinksma (Subsection 4.1.5) provide a strategy to deal with these cases.

In the third table (Table 4.3), the following characteristics are considered: (1) the kind of time (analogue or digital-time); (2) the kind of test cases (instantiated or abstract); (3) the kind of communication allowed by models; (4) the kind of oracle (manual, partial, or automated). Considering the first characteristic of Table 4.3, almost all approaches adopt the analogue-time model and represent time in a symbolic way. An interesting characteristic of Krichen and Tripakis' work is that they do not represent time symbolically, but they provide

Work	Conf. Relation	Specification	Implementation
Cardell-Oliver	trace equivalence	input-complete and must have more states than the implementation.	input-complete
En-Nouaary et al.	trace equivalence	input-complete and must have the same number of locations as the implementation.	input-complete
Li et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Khoumsi	timed trace inclusion	input-complete	input-complete
Briones and Brinksma	ioco with quiescence	input-complete	input-complete*
Bohnenkamp and Belinfante	ioco with quiescence	input-complete	input-complete
Bodeveix et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Larsen et al.	timed trace inclusion	deterministic and input-complete	input-complete
Hessel et al.	timed trace inclusion	deterministic, input-complete, and output urgent	input-complete
Merayo et al.	there are several conformance relations	input-complete	input-complete
Krichen and Tripakis	tioco	no restriction on input-completeness	input-complete
Zheng et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
David et al.	tioco	input-complete	input-complete
Adjir et al.	timed trace inclusion	deterministic, input-complete, and output urgent	deterministic, input-complete, and output urgent
Styp et al.	stioco	non-deterministic	input-complete
Timo et al.	tvco	assumptions are not discussed	assumptions are not discussed

Tabela 4.2: Related Work

a symbolic reachability algorithm to generate tests. Timo et al. (Subsection 4.1.16) propose a reachability analysis based on clock regions, but this strategy can quickly lead to the state space explosion problem. It is important to remark that complete real-time symbolic testing strategies that take system variables, parameters and time into account are nonexistent.

Almost all approaches generate instantiated test cases, since only time is abstracted during the test generation. Considering the approaches proposed by Styp et al. (Subsection 4.1.15) and Timo et al. (Subsection 4.1.16), it is not possible to define the kind of test case because neither examples are presented nor test cases are formally defined. Considering the possibility of specification of communicating elements, all approaches only allow synchronous communication. Thus, it is not possible to model asynchronous events such as interruptions. Considering the last characteristic in Table 4.3, most approaches only provided test case generation algorithms, but in the context of real-time systems the execution of test cases and verdicts assignment are as difficult as the generation of tests. Only three approaches developed an automated oracle. Basically, Bohnenkamp and Belinfante (Subsection 4.1.6), Larsen et al. (Subsection 4.1.8), and Krichen and Tripakis (Subsection 4.1.11) developed algorithms that use the specification to guide the execution of tests and assignment of verdicts. When the specification has only actions without parameters and variables the development of automated oracles is relatively simple. Nevertheless, an automated oracle in a real-time symbolic testing strategy causes the test data generation problem because variables and parameters must be instantiated during the test execution.

4.3 Problem Statements

This section describes several problems identified during the review of the work related to this thesis. A practical example adapted from [110] will be used to clarify the discussion. The chosen example is a burglar alarm system, a real-time monitoring system. The objective of the system is to monitor sensors to detect the presence of intruders in a building.

This system uses different kinds of sensors including movement detectors in individual rooms, window sensors, which detect the breaking of a window and door sensors, which detect the opening of doors. There are 50 window sensors, 30 door sensors, and 200 movement detectors. When a sensor indicates the presence of an intruder, the system automatically

Work	Time	Test Cases	Communication	Oracle
Cardell-Oliver	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
En-Nouaary et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Li et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Khoumsi	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Briones and Brinksma	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Bohnenkamp and Belinfante	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Bodeveix et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Larsen et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Hessel et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Merayo et al.	digital-time model	instantiated	synchronous	partial
Krichen and Tripakis	digital and analogue-time models*	instantiated	synchronous	automated
Zheng et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
David et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Adjir et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Styp et al.	analogue-time model	undefined	synchronous	undefined
Timo et al.	analogue-time model	undefined	synchronous	undefined

Tabela 4.3: Related Work

calls the police and, with a voice synthesiser, reports the position of the alarm. In addition, the system switches on lights around the area with activated sensors and switches on an audible alarm. The system is normally powered by the central power supply system, but it is equipped with a battery backup. The loss of power is detected by a circuit monitor that monitors the main tension. The system switches automatically to backup power when a voltage drop is detected. The timing requirements contained in our version of the burglar alarm system are described in Table 4.4.

Stimulus/Response	**Timing Requirements**
Power fail interrupt	The switch to backup power must be completed within a deadline of 50 ms.
Audible alarm	The audible alarm should be switched on within 1/2 second after an alarm is raised by a sensor.
Voice synthesiser	A synthesised message should be available within 3 seconds after an audible alarm is switched on.
Communications	The call to the police should be started within 1 second after a message is synthesised.
Lights switch	The lights should be switched on within 1/2 second after the calling to the police.

Tabela 4.4: Timing Requirements

Considering the architecture of the system, each system functionality is allocated to a concurrent process as well as each kind of sensor is allocated to a process. There is an interruption-driven system to deal with the failure and switching of power supply, a communication system, a voice synthesiser, an audible alarm system, and an illumination drive system to turn on lights around the sensor. The architecture of the system is depicted in Figure 4.1. The labelled arrows indicate the data flow between processes and the notes associated with the processes indicate which process or action causes the interruption.

Considering the specified scenario, some limitations of the approaches presented in this chapter are discussed. As an alarm system is naturally a system that has several quiescent states it is important to consider this property during the testing process. In this sense, the first problem that arises is the lack of tools for testing quiescence in real-time systems.

Another limitation of the current work is the input-completeness assumption of implementations. As discussed in Section 4.2, there are situations where some inputs can be forbidden or ignored by the system. These cases are only considered by Briones and Brinksma [24], but all forbidden or ignored inputs must be previously specified. Thus, there are still

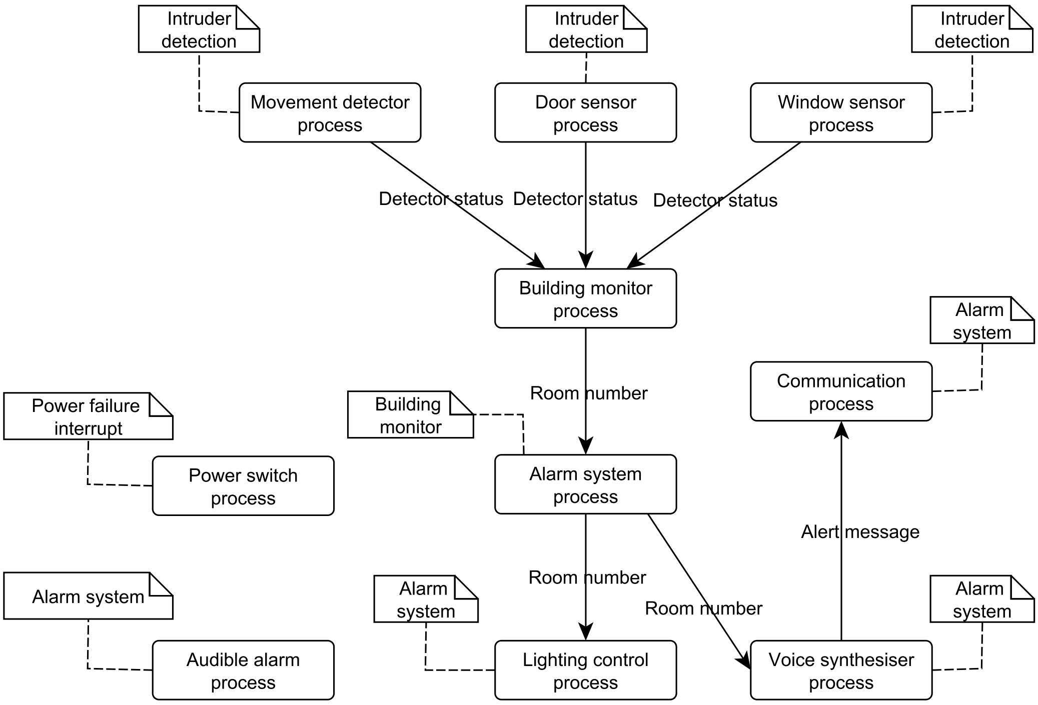


Figura 4.1: Burglar Alarm System Architecture

open problems such as the cases where a system cannot accept an input because of a fault (e.g. a requirement was implemented but there is not an option menu to access it). This problem is very common in the test of mobile phone applications and it is present not only in the real-time systems context but also in non-real-time systems.

The most efficient implementation of the burglar alarm system is to adopt the interrupt-driven architecture of Figure 4.1 where most communication between processes is asynchronous. For example, when an intruder is detected, the process that controls the activated sensors interrupts the building monitor process, which interrupts the alarm system process. Finally, the alarm system process, using interruptions, activates the following processes: audible alarm, lighting control, voice synthesiser, and communication process. However, current real-time models do not take asynchronous events into account. Thus, the testing process of real-time systems with asynchronous events based on formal models is compromised, if not impossible.

As discussed in Section 4.2, all approaches only use symbolic strategies to abstract time as a way of digitisation of analogue-time models. Nevertheless, when the system uses huge data domains, each value continues to be represented as a system state, leading to the classical state space explosion problem. Moreover, the digitisation of analogue-time models usually leads to a huge number of test cases. An interesting solution would be to provide symbolic testing strategies to abstract not only time but also variables of the system, leading to a simplified test suite where only most significant system data would be used during the test execution.

Taking the burglar alarm system as an example, the representation of the three kinds of sensors (movement, door and window sensors) using the existing real-time models would lead to three different paths in the model. In a real-time symbolic model, the representation is simple: the processes representing the three kinds of sensors could be abstracted in only one location in the model, where the interruption action would carry the information about the kind of sensor and the room number as parameters. In addition, the number of rooms of the burglar alarm system could be abstracted in a variable, thus it would be simpler to model situations where depending on the number of activated sensors, the system could take different decisions.

The high abstraction level in symbolic models leads to other problems such as the oracle problem. As it is possible to generate abstract test cases, the oracle problem is related to symbolic strategies in the sense that it is more difficult to provide an automated way for test case generation and execution, since test cases must be instantiated according to constraints defined in the specification.

4.4 Concluding Remarks

This chapter presented a review of work related to this thesis and several problems were stated. It is not our intention to deal with all identified problems. As discussed in Chapter 1, we intend to propose an extension of the symbolic testing strategy presented in Subsection 2.1.9 to deal with real-time systems. Furthermore, we intend to provide ways of modelling and testing asynchronous events considering an automated oracle for test case generation and execution. Problems related to quiescence and input-completeness are outside the scope of this thesis.

Capítulo 5

Timed Input-Output Symbolic Transition Systems

Este capítulo apresenta um novo modelo simbólico chamado *Timed Input-Output Symbolic Transition System* (TIOSTS) [11]. O objetivo é tratar as limitações dos formalismos existentes abstraindo tempo e dados durante a geração de casos de teste. Este modelo é uma extensão de dois modelos existentes: *Timed Automata with Inputs and Outputs* (TAIO), que por sua vez são uma extensão de *Timed Automata* [5] com entradas e saídas associadas a prazos para modelar urgência [21], e *Input-Output Symbolic Transition Systems* (IOSTS) [108]. Em outras palavras, um TIOSTS é um autômato com um conjunto finito de nós, variáveis utilizadas para representar dados do sistema e um conjunto finito de relógios utilizados para representar a evolução do tempo. Uma transição é composta por uma condição envolvendo variáveis e relógios, uma ação com parâmetros para a comunicação com o ambiente, atribuições a variáveis e reinicialização de relógios.

5.1 Syntax of TIOSTS

We intuitively explain the different notions of the TIOSTS model through the example shown in Figure 5.1[1] that models a withdrawal transaction in an ATM system. In a TIOSTS, a transition is fired if its guard is true, then the action is executed and all assignments are

[1] In graphical representations, input actions are followed by the "?" symbol and output actions are followed by the "!" symbol. These symbols are used only as visual notation, they are not part of the action's name.

performed.

The withdrawal transaction has a precondition (an initial condition) that states that the current *balance* must be strictly positive. Initially, the system is in the *Idle* location where it expects the *Withdrawal* input carrying a strictly positive integer parameter *amount* that is saved into the *withdrawalValue* variable with the clock set to zero when the transition is taken. The scope of an action parameter is local with respect to the transition where it appears, thus the value of an action parameter must be stored in a variable in order to use it in the future.

Considering that the value of *withdrawalValue* is less than or equal to the *balance* and the time represented by *clock* is less than or equal to 10 time units, the ATM system dispenses the cash through the *DispenseCash* output carrying the *amount* parameter (the condition $amount = withdrawalValue$ contained in the guard means "choose a value for the *amount* parameter that, with the value of the *withdrawalValue* variable, satisfies the guard"). This is a characteristic inherited from IOSTS models whose objective is to associate the value of a variable with an action parameter in order to define output actions. Finally, the balance variable is decreased by the withdrawn value, and the system returns to the *Idle* location.

On the other hand, if the account does not have sufficient funds, the system must emit the invalid withdrawal value through the *InsufficientFunds* output carrying the *amount* parameter when *clock* is at most 2 (the condition $amount = withdrawalValue$ has a similar meaning to the previous guard), and reset the clock to zero again. Finally, the current balance is emitted through the *PrintBalance* output when *clock* is at most 5 (the condition $amount = balance$ contained in the guard means "choose a value for the *amount* parameter such that it is equal to the value of the *balance* parameter"), and the system returns to the *Idle* location.

Guards on transitions only indicate when they are enabled or not, but they cannot force the transition to be taken. Considering the specification of Figure 5.1, the TIOSTS may stay forever in any location. This can be solved by adding some restrictions to the transitions in order to describe the urgency of execution. Adopting the strategy defined in [21], each transition is annotated with one of the following three deadlines: *lazy*, *delayable*, and *eager*. The *lazy* deadline imposes no urgency to the transition to be taken, *delayable* means that once enabled the transition must be taken before it becomes disabled, and *eager* means the transition must be taken as soon as it becomes enabled.

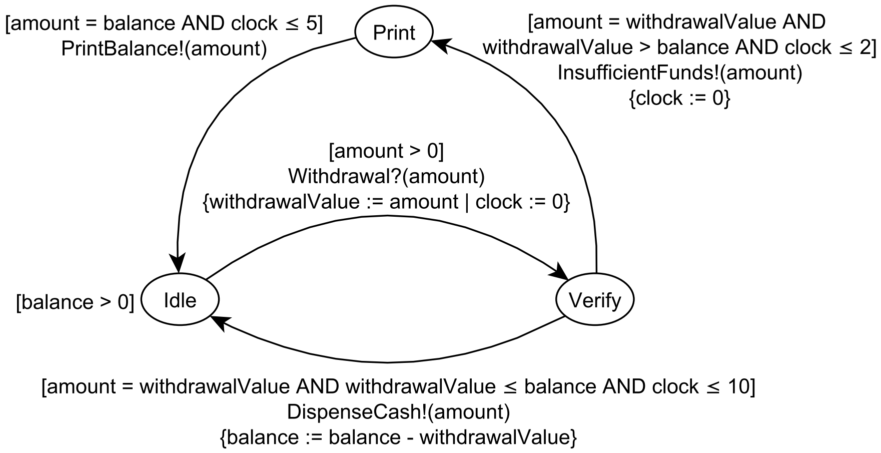

[amount = balance AND clock ≤ 5]
PrintBalance!(amount)
Print
[amount = withdrawalValue AND
withdrawalValue > balance AND clock ≤ 2]
InsufficientFunds!(amount)
{clock := 0}
[amount > 0]
Withdrawal?(amount)
{withdrawalValue := amount | clock := 0}
[balance > 0]
Idle
Verify
[amount = withdrawalValue AND withdrawalValue ≤ balance AND clock ≤ 10]
DispenseCash!(amount)
{balance := balance - withdrawalValue}

Figura 5.1: TIOSTS Example

Default deadlines are adopted in order to not overload pictures, thus when not specified the deadline of transitions with output actions is assumed to be *delayable* and the deadline of transitions with input actions is assumed to be *lazy*. On the other hand, when different deadlines are necessary they must be explicitly specified. A TIOSTS is formally described in Definition 5.1.

Definition 5.1 (TIOSTS). *Formally, a TIOSTS is a tuple $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$, where:*

- *V is a finite set of typed variables;*
- *P is a finite set of parameters. For $x \in V \cup P$, $type(x)$ denotes the type of x;*
- *Θ is the initial condition, a predicate with variables in V;*
- *L is a finite, non-empty set of locations;*
- *$l^0 \in L$ is the initial location;*
- *$\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is a non-empty, finite alphabet, which is the disjoint union of a set $\Sigma^?$ of input actions, a set $\Sigma^!$ of output actions, and a set Σ^τ of internal actions. Each action $a \in \Sigma$ has a signature $sig(a) = \langle p_1, ..., p_n\rangle$, that is a tuple of distinct parameters. The signature of internal actions is the empty tuple;*
- *C is a finite set of clocks;*
- *$\mathcal{T}$ is a finite set of transitions. Each transition $t \in \mathcal{T}$ is a tuple $\langle l, a, G, A, y, l'\rangle$, where:*

- *$l \in L$ is the origin location of the transition,*

- *$a \in \Sigma$ is the action,*

- *$G = G^D \wedge G^C$ is the guard, where G^D is a predicate over variables in $V \cup sig(a)$*[2] *and G^C is a clock constraint over C defined as a conjunction of constraints of the form $\alpha \# c$, where $\alpha \in C$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$,*

- *$A = A^D \cup A^C$ is the assignment of the transition. For each variable $x \in V$ there is exactly one assignment in A^D, of the form $x := A^{D^x}$, where A^{D^x} is an expression on $V \cup sig(a)$. $A^C \subseteq C$ is the set of clocks to be reset,*

- *$y \in \{$lazy, delayable, eager$\}$ is the deadline of the transition,*

- *$l' \in L$ is the destination location of the transition.*

◇

5.2 Semantics of TIOSTS

TLTS and TIOLTS models are used to define the semantics of all approaches based on TA and TAIO, respectively. Thus, the semantics of a TIOSTS $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ is described in terms of a TIOLTS (Definition 5.2). Intuitively, the TIOLTS states expand the sets of locations, of *valuations* of variables V and clocks C, while transitions expand the sets of actions Σ associated with parameter values P. A *valuation* of the variables in V is a mapping ν which maps every variable $x \in V$ to a value $\nu(x)$ in the domain of x. Valuations of parameters P are defined similarly. Let $\mathcal{V}$ denote the set of valuations of the variables V and let Γ denote the set of valuations of the parameters P. Let the function $\psi : C \rightarrow \mathbb{R}^{\geq 0}$ denote a clock valuation. We denote by $\overline{0}$ the valuation that assigns 0 to all clocks.

Considering $\nu \in \mathcal{V}$ and $\gamma \in \Gamma$, for an expression E involving a subset of $V \cup P$, we denote by $E(\nu, \gamma)$ the value obtained by evaluating the result of substituting in E each variable by its value according to ν and each parameter by its value according to γ.

Definition 5.2 (TIOLTS semantics of a TIOSTS)**.** *The semantics of a TIOSTS $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ is a TIOLTS $[\![W]\!] = \langle S, S^0, Act, T \rangle$, defined as follows:*

[2] G^D is assumed to be expressed in a theory in which satisfiability is decidable.

- *$S = L \times \mathcal{V} \times (C \rightarrow \mathbb{R}^{\geq 0})$ is the set of states of the form $s = \langle l, \nu, \psi \rangle$ where $l \in L$ is a location, $\nu \in \mathcal{V}$ is a specific valuation for all variables V, and ψ is a clock valuation;*

- *$S^0 = \{\langle l^0, \nu, \psi \rangle \mid \Theta(\nu) = true,\ \overline{0}\}$ is the set of initial states. It is important to remark that the number of initial states can be infinite because, in this case, there may be infinite valuations satisfying the initial condition Θ;*

- *$Act = \Lambda \cup D$ is the set of actions, where $\Lambda = \{\langle a, \gamma \rangle \mid a \in \Sigma, \gamma \in \Gamma_{sig(a)}\}$ is the set of discrete actions and $D = \mathbb{R}^{\geq 0}$ is the set of time-elapsing actions. Λ is partitioned into the sets $\Lambda^?$ of input actions, $\Lambda^!$ of output actions, and Λ^τ of internal actions;*

- *T is the transition relation defined as follows: (1) transitions with discrete actions are of the form $\langle l, \nu, \psi \rangle \overset{\langle a,\gamma \rangle}{\rightarrow} (l', \nu', \psi')$, where the system moves from $\langle l, \nu, \psi \rangle$ to $\langle l', \nu', \psi' \rangle$ through an action $\langle a, \gamma \rangle$ if there is a transition $t : \langle l, a, G, A, y, l' \rangle \in \mathcal{T}$ such that G evaluates to $true$, $\nu' = A^D(\nu, \gamma)$, and $\psi' = A^C(\psi)$; (2) transitions with time-elapsing actions are of the form $(l, \nu, \psi) \overset{d}{\rightarrow} (l, \nu, \psi + d)$ for all $d \in D$ considering that the deadlines do not block time progress. Once the lazy deadline is used only to denote the absence of deadlines, lazy transitions cannot block time progress. A delayable transition can block time progress if there exist $0 \leq d_1 < d_2 \leq d$ such that $\psi + d_1 \models G^C$ and $\psi + d_2 \not\models G^C$, whereas an eager transition can block time progress if $\psi \models G^C$.*

◇

As in [78], delayable transitions with guards of the form $\alpha < c$ are not allowed because there is no latest time so that the guard is still true. Also, eager transitions with guards of the form $\alpha > c$ are not allowed because there is no earliest time so that the guard becomes true.

Most notions and properties of TIOSTS are defined in terms of their underlying TIOLTS semantics (Definition 5.2). Then, consider $s, s', s_i \in S$; $\tau_i \in \Lambda^\tau$; $\omega, \omega_i \in Act$; and $a, a_i \in (Act \backslash \Lambda^\tau)$. Moreover, let $\rho \in Act^*$ be a sequence of discrete actions and time-elapsing actions, and $\sigma \in (Act \backslash \Lambda^\tau)^*$ be a sequence of visible discrete and time-elapsing actions. $\epsilon \in Act^*$ is the empty sequence. The sum of all delays spent in a sequence of actions ρ (respectively σ) is denoted by time(ρ) (respectively by time(σ)). For example, time(ϵ) $= 0$ and time(2.5 a? 0.5 x!) $= 3.0$.

Let $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ be a TIOSTS whose semantics is defined by the TIOLTS $[\![W]\!] = \langle S, S^0, \mathit{Act}, T \rangle$. We write $s \xrightarrow{\omega} s'$ for $(s, w, s') \in T$, $s \xrightarrow{\omega}$ for $\exists s' : s \xrightarrow{\omega} s'$. Let $s \xrightarrow{\omega_1 ... \omega_n} s' \triangleq \exists s_0, ..., s_n : s = s_0 \xrightarrow{\omega_1} s_1 \xrightarrow{\omega_2} \cdots \xrightarrow{\omega_n} s_n = s'$ be an *execution*. We also write $s \xrightarrow{\rho}$ for $\exists s' : s \xrightarrow{\rho} s'$. *Traces*$(s) \triangleq \{\rho \in \mathit{Act}^* \mid s \xrightarrow{\rho}\}$ describes the set of sequences of discrete and time-elapsing actions fireable from s. The set of fireable actions from s is defined by $\Omega(s) \triangleq \{\omega \in \mathit{Act} \mid s \xrightarrow{\omega}\}$. $Out(s) \triangleq \Omega(s) \cap (\Lambda^! \cup D)$ is the set of all output events (including time-elapsing actions) fireable from s. The definition of $Out(s)$ can be extended for sets of states: for $P \subseteq S$ we have $Out(P) \triangleq \bigcup_{s \in P} Out(s)$.

The $\Rightarrow$ relation is used to denote the observable behaviour. Given $s, s' \in S$, $d \in \mathbb{R}^{\geq 0}$ and $a \in \Lambda^! \cup \Lambda^?$, we have $s \overset{d}{\Rightarrow} s'$ whenever $\exists \rho \in (\Lambda^\tau \cup D)^*$ such that $s \xrightarrow{\rho} s'$ and $time(\rho) = d$, whereas we have $s \overset{a}{\Rightarrow} s'$ whenever $\exists \rho_1, \rho_2 \in (\Lambda^\tau)^*$, $s_1, s_2 \in S$ such that $s \xrightarrow{\rho_1} s_1 \xrightarrow{a} s_2 \xrightarrow{\rho_2} s'$. Given $a_1, \cdots a_n \in (\mathit{Act} \backslash \Lambda^\tau)^*$, an *observable execution* is defined as $s \overset{a_1 ... a_n}{\Rightarrow} s' \triangleq \exists s_0, ..., s_n : s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} s_n = s'$. For $a \in \mathit{Act} \setminus \Lambda^\tau$ we also define $s \overset{a}{\Rightarrow} \triangleq \exists s' : s \overset{a}{\Rightarrow} s'$ and for $\sigma \in (\mathit{Act} \setminus \Lambda^\tau)^*$, $s \overset{\sigma}{\Rightarrow} \triangleq \exists s' : s \overset{\sigma}{\Rightarrow} s'$. *ObservableTraces*$(s) \triangleq \{\sigma \in (\mathit{Act} \setminus \Lambda^\tau)^* \mid s \overset{\sigma}{\Rightarrow}\}$ describes the set of sequences of observable and time-elapsing actions fireable from s. Finally, the set of sequences of observable behaviours fireable from the initial state of a TIOSTS W is defined by *ObservableTraces*$(W) \triangleq$ *ObservableTraces*(S_0).

The set s *after* $\sigma \triangleq \{s' \in S \mid s \overset{\sigma}{\Rightarrow} s'\}$ is the set of states reachable from s after the execution of σ, and P *after* $\sigma \triangleq \bigcup_{s \in P} s$ *after* σ is the set of states reachable from the set P after the execution of σ.

Subclasses of TIOSTS. Let $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ be a TIOSTS and $[\![W]\!] = \langle S, S^0, \mathit{Act}, T \rangle$ its associated TIOLTS. W is *complete* if it can accept any action at any state, i.e., $\forall s \in S, b \in \Lambda : s \xrightarrow{b}$. On the other hand, W is *input-complete* if it can accept any input action at any state, possibly after internal actions, i.e., $\forall s \in S, b \in \Lambda^? : s \overset{b}{\Rightarrow}$. W is said to be a *lazy-action* TIOSTS if the deadlines of all transitions are lazy, that is, $\mathcal{T} = \{t \mid t : \langle l, a, G, A, lazy, l' \rangle \in \mathcal{T}\}$. W is said to be a *non-blocking* TIOSTS when it does not block time. In this case, the following condition must be satisfied [74]: $\forall s \in S_0$ *after* ρ, $\forall d \in \mathbb{R}^{\geq 0}$, $\exists \rho' \in (\Lambda^! \cup \Lambda^\tau \cup D)^* : \text{time}(\rho') = d \wedge s \xrightarrow{\rho'}$. W is said to be *deterministic* if the following three conditions are satisfied [35; 63; 108]:

1. $\Lambda^{\tau} = \emptyset$ (i.e. there are no internal actions);

2. $\mid S_0 \mid = 1$, that is, there is only one initial state implying the initial condition Θ is satisfied by only one valuation ν^0;

3. for all $l \in L$ and for each pair of distinct transitions with origin in l carrying the same action a, that is, $t_1 : \langle l, a, G_1, A_1, y_1, l'_1 \rangle$ and $t_2 : \langle l, a, G_2, A_2, y_2, l'_2 \rangle$, the guards G_1 and G_2 are mutually exclusive (i.e., $G_1 \wedge G_2$ is unsatisfiable).

5.3 Synchronous Product of TIOSTS

The synchronous product of two TIOSTSs W_1 and W_2 is an important operation used in both property oriented testing and conformance testing. This operation is used in the former for identifying behaviours of the specification accepted or rejected by a particular property (e.g., W_1 could be a specification and W_2 could be a test purpose). On the other hand, for conformance testing, this operation is used for modelling the synchronous execution of a test case on an implementation (e.g., W_1 could be a test case and W_2 could be an implementation under test). This classical problem is known as the language intersection problem [62].

The synchronous product operation requires compatibility between W_1 and W_2, that is, W_1 and W_2 must share the same sets of input and output actions from the same signature, with the same set of parameters, and have no variables, internal actions, or clocks in common.

Definition 5.3 (Compatibility for Synchronous Product)**.** *The TIOSTSs* $W_i = \langle V_i, P_i, \Theta_i, L_i, l_i^0, \Sigma_i, C_i, \mathcal{T}_i \rangle$ *(*$i = 1, 2$*) are compatible if* $V_1 \cap V_2 = \emptyset, P_1 = P_2, \Sigma_1^? = \Sigma_2^?, \Sigma_1^! = \Sigma_2^!, \Sigma_1^{\tau} \cap \Sigma_2^{\tau} = \emptyset$*, and* $C_1 \cap C_2 = \emptyset$*.* $\diamond$

Given the ordering $lazy < delayable < eager$ on deadlines and two deadlines y_1, y_2, $op(y_1, y_2) = (y_2$ if $y_1 < y_2$ and y_1 otherwise) is an operation which computes the resulting deadline in the synchronous product operation by keeping the most restrictive one.

Given two compatible TIOSTSs, Definition 5.4 formally describes the synchronous product between them.

Definition 5.4 (Synchronous Product)**.** *The synchronous product of two compatible TIOSTSs* W_1 *and* W_2 *is denoted by* $SP = W_1 \parallel W_2$*.* SP *is the TIOSTS* $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$

defined by: $V = V_1 \cup V_2, P = P_1 = P_2, \Theta = \Theta_1 \wedge \Theta_2, L = L_1 \times L_2, l^0 = \langle l_1^0, l_2^0 \rangle, \Sigma^? = \Sigma_1^? = \Sigma_2^?, \Sigma^! = \Sigma_1^! = \Sigma_2^!, \Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$, *and* $C = C_1 \cup C_2$. *The set* $\mathcal{T}$ *is the smallest set such that:*

1. *For* $a \in \Sigma_1^\tau$ *and* $l_2 \in L_2$*:*

$$\text{if } \langle l_1, a, G_1, A_1, y_1, l_1' \rangle \in \mathcal{T}_1 \text{ then } \langle \langle l_1, l_2 \rangle, a, G_1, A_1, y_1, \langle l_1', l_2 \rangle \rangle \in \mathcal{T};$$

2. *For* $a \in \Sigma_2^\tau$ *and* $l_1 \in L_1$*:*

$$\text{if } \langle l_2, a, G_2, A_2, y_2, l_2' \rangle \in \mathcal{T}_2 \text{ then } \langle \langle l_1, l_2 \rangle, a, G_2, A_2, y_2, \langle l_1, l_2' \rangle \rangle \in \mathcal{T};$$

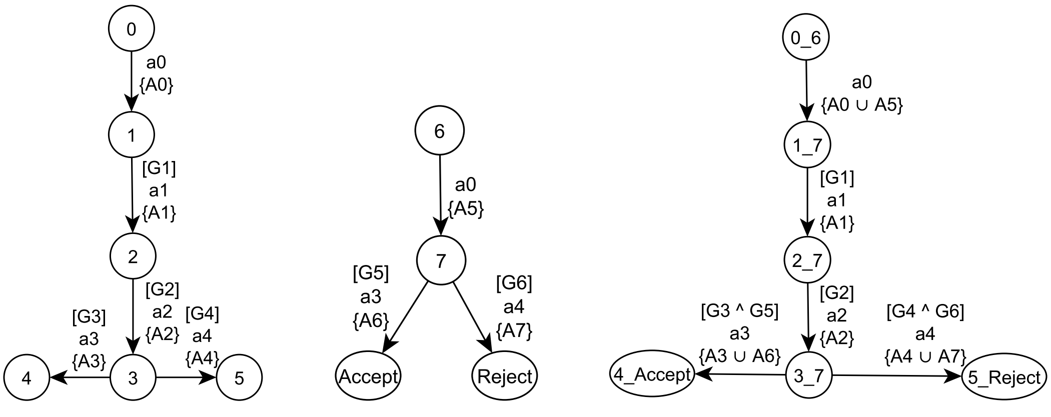
3. *For* $a \in \Sigma^? \cup \Sigma^!$*:*

$$\text{if } \langle l_1, a, G_1, A_1, y_1, l_1' \rangle \in \mathcal{T}_1 \text{ and } \langle l_2, a, G_2, A_2, y_2, l_2' \rangle \in \mathcal{T}_2 \text{ then}$$

$$\langle \langle l_1, l_2 \rangle, a, G_1 \wedge G_2, A_1 \cup A_2, op(y_1, y_2), \langle l_1', l_2' \rangle \rangle \in \mathcal{T}.$$

◇

Considering the Definition 5.4, the execution of internal actions can occur independently and it is described by Rules 1 and 2. Rule 3 describes the synchronization of W_1 and W_2 through observable actions. Figure 5.2 presents an example of the synchronous product between a TIOSTS W_1 (Figure 5.2(a)) and a TIOSTS W_2 (Figure 5.2(b)), obtaining the TIOSTS of Figure 5.2(c) as result.



(a) TIOSTS W_1 (b) TIOSTS W_2 (c) $SP = W_1 \parallel W_2$

Figura 5.2: Synchronous Product Example

5.4 Concluding Remarks

This chapter presented the symbolic model proposed to address limitations of the existing notations abstracting time and data in the specification of real-time systems. As the proposed model is based on timed automata, it has the same expressiveness as the approaches presented in Chapter 4. However, the use of variables to represent the system data leads to more compact and abstract models. Moreover, when data and time are treated in a symbolic way the state space explosion problem is avoided. Next chapter presents how test cases can be generated from the proposed model.

Capítulo 6

Teste de Conformidade com TIOSTS

Este capítulo instancia o arcabouço de teste de conformidade apresentado na Subseção 2.1.5 e o arcabouço de propósito de teste apresentado na Subseção 2.1.6 considerando os modelos TIOSTS definidos no Capítulo 5. Em seguida, o processo de geração de casos de teste baseado em modelos TIOSTS definido com a finalidade de checar a conformidade entre uma especificação e uma implementação é descrito. Finalmente, algumas propriedades dos casos de teste gerados através da nossa abordagem são discutidas. O conteúdo deste capítulo também está descrito em [11].

6.1 Testing Conformance

Conformance testing is a kind of testing used to ensure that an implementation of a software system meets its specification [118]. This kind of testing relates a specification with an implementation through a conformance relation, which is checked by the execution of test cases, possibly selected according to a test purpose. Thus, it is essential to describe all concepts related to conformance testing such as specifications, implementations, conformance relations between specifications and implementations, and test cases.

Specifications. A specification is a formal model of the SUT represented by a non-blocking TIOSTS $\mathcal{S}$. The non-blocking specification assumption is due to the fact that we are considering specifications of software systems that do not force input actions, that is, the system cannot block because an input action was not provided by the environment.

Implementations. An implementation is a physical software system running on a real-time environment (e.g., a real-time operating system). In order to reason about conformance, it is assumed that the semantics of any implementation can be modelled by a formal object. We assume here that it is modelled by a TIOLTS $\mathcal{I}$. Moreover, the implementation is assumed to be input-complete, non-blocking, and has the same interface (input and output actions with their signatures) as the specification $\mathcal{S}$. These assumptions are called *test hypotheses*.

Test Cases. Test cases (Definition 6.1) are used to check the conformance between specifications and implementations. It is here defined as a TIOSTS TC as follows:

Definition 6.1 (Test Case)**.** *A test case is a deterministic, input-complete TIOSTS* $TC = \langle V_{TC}, P_{TC}, \Theta_{TC}, L_{TC}, l^0_{TC}, \Sigma_{TC}, C_{TC}, \mathcal{T}_{TC}\rangle$*, equipped with three disjoint sets of locations* Pass*,* Fail*, and* Inconclusive*. Moreover, the set of actions is* $\Sigma_{TC} = \Sigma^?_{TC} \cup \Sigma^!_{TC}$*, where* $\Sigma^?_{TC} = \Sigma^!_{\text{SUT}}$ *(outputs of the* SUT *are the inputs of the TC) and* $\Sigma^!_{TC} = \Sigma^?_{\text{SUT}}$ *(TC emits only inputs allowed by the* SUT*).* ◇

Intuitively, when the location $Fail$ is reached, it means rejection, the location $Pass$ means that some targeted behaviour has been reached (this will be clarified later) and $Inconclusive$ means that targeted behaviours cannot be reached anymore.

Conformance Relation. The conformance relation considered is the **tioco** relation defined by Krichen and Tripakis in [76; 77]. Informally, an implementation conforms to a specification for **tioco** if and only if, after any trace of the specification, any output action (including time-elapsing actions) that the implementation provides after this trace is an output action that the specification may also provide.

Definition 6.2 (tioco)**.** *An implementation* $\mathcal{I}$ *conforms to a specification* $\mathcal{S}$ *for* ***tioco****, denoted by* $\mathcal{I}$ ***tioco*** $\mathcal{S}$*, iff* $\forall\sigma \in \text{ObservableTraces}(\mathcal{S}),\ Out(\mathcal{I}\ \textit{after}\ \sigma) \subseteq Out(\mathcal{S}\ \textit{after}\ \sigma)$. ◇

6.2 Test Case Generation Process

The test case generation process derives test cases from specifications according to the conformance relation. For simplicity, we shall assume that the specification $\mathcal{S}$ is deterministic and non-blocking. However, it is possible to deal with non-determinism, under some assumptions, for both data [65] and time [17]. It is important to remark that internal actions, quiescence, and non-input-completeness of implementations are not considered in the proposed test case generation process because these characteristics are outside the scope of this thesis. The proposed process considers the selection of test cases by test purposes.

Test Purposes. A test purpose describes some desired behaviours that we wish to check on the implementation during the test campaign. They are used to select test cases in order to check specific scenarios. In our setting, a test purpose is a particular TIOSTS TP formally described as follows:

Definition 6.3 (Test Purpose). *Given a specification TIOSTS $\mathcal{S}$ with action alphabet Σ, a test purpose is a deterministic, complete, lazy-action TIOSTS $TP = \langle V_{TP}, P_{TP}, \Theta_{TP}, L_{TP}, l^0_{TP}, \Sigma_{TP}, C_{TP}, \mathcal{T}_{TP} \rangle$, equipped with a special set of locations* Accept $\subseteq L_{TP}$ *such that all transitions leaving these locations are self-loops*[1]. *Moreover TP has to be compatible with $\mathcal{S}$ thus $\Sigma_{TP} = \Sigma$.* ◇

The selection is performed through the synchronous product operation defined in Section 5.3. For this, complete test purposes are needed to ensure that the runs of a specification are not restricted before they are accepted (if ever).

Accept locations are used to indicate that the expected scenario modelled by the test purpose has been fulfilled, while *Reject* locations are used otherwise. Figure 6.1 presents an example of a test purpose for the withdrawal transaction example presented in Subsection 5.1. It is used to select the scenarios where the user successfully performs a withdrawal transaction. The *Reject* location is used to discard all other scenarios where the system does not exhibit the desired behaviour.

It is important to note that this test purpose is not complete (i.e., not all actions are enabled at any location), but using a strategy defined in [108] it is possible to automatically complete it. The steps to automatically complete a test purpose are: (1) in each location, add a self-loop with an action not enabled; (2) for each transition with a guard G and an action a, create a new transition to the *Reject* location with the same action a and the negation of

[1] One can also consider another set of locations $Reject$ that can be used to discard all other scenarios where the system does not exhibit the desired behaviour.

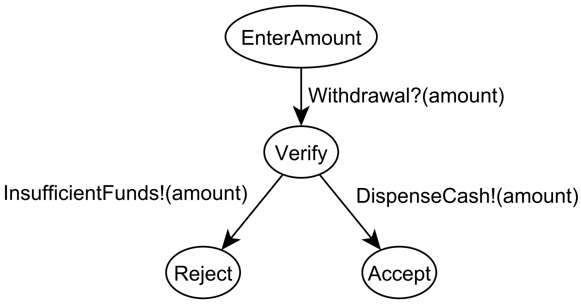


Figura 6.1: TIOSTS Test Purpose Example

the conjunction of all guards associated with a. With this automatic operation the activity of defining test purposes is simplified by allowing the tester to focus only on the desired behaviour.

The test case generation process starts with the specification $\mathcal{S}$ of the SUT $\mathcal{I}$ and a test purpose TP. Test purposes are used in order to verify if the SUT exhibits a desired behaviour and their definition allow the tester to focus only on specific behaviours. In this case, test purposes need to be completed because all input actions are not enabled all the time. The specification of $\mathcal{I}$ is combined with the completed test purpose through the computation of the synchronous product (Definition 5.4). Then, the resulting TIOSTS model is symbolically executed to identify and select possible traces leading to an *Accept* location. Finally, the selected trace is translated into a test case considering the TIOSTS notation. A general view of this test process is presented in Figure 6.2. Each step of this process is detailed in the remainder of this section.

6.2.1 Test Purpose Completion

Algorithm 6.1 presents a simplified implementation of the test purpose completion operation. This algorithm requires only one parameter: the test purpose to be completed.

During the algorithm execution all non-verdict locations of the test purpose are analysed (Lines from 2 to 4). If some location has not enabled actions, then a self loop is created with these actions (Lines from 5 to 14).

After that, for each outgoing transition of the location being processed that has any guard, it is created a new transition without assignments and as guard the negation of the conjunction of all guards associated with the action of the current transition (Lines from 15 to 22). If

Algorithm 6.1: Test Purpose Completion Algorithm

```
complete(TIOSTS $TP$){
  Set locations := $TP$.getLocations();
  for (Location location : locations) {
    if (!isVerdict(location)) {
      for (Action action : remaningActions(location)) {
        Transition transition := new Transition();
        transition.setSource(location);
        transition.setGuard(true);
        transition.setAction(action);
        transition.setAssignments($\emptyset$);
        transition.setDeadline(lazy);
        transition.setTarget(location);
        $TP$.addTransition(transition);
      }
      for (Transition t : location.getOutGoingTransitions()) {
        if (!t.getGuard().isEmpty()) {
          Transition transition := new Transition();
          transition.setSource(location);
          transition.setGuard(
              negation(location.getAllGuards(t.getAction())));
          transition.setAction(t.getAction());
          transition.setAssignments($\emptyset$);
          transition.setDeadline(lazy);
          if (t.getTarget() = reject) {
            transition.setTarget(location);
          } else {
            transition.setTarget(reject);
          }
          if (!$TP$.contains(transition)) {
            $TP$.addTransition(transition);
          }
        }
      }
    }
  }
}
```

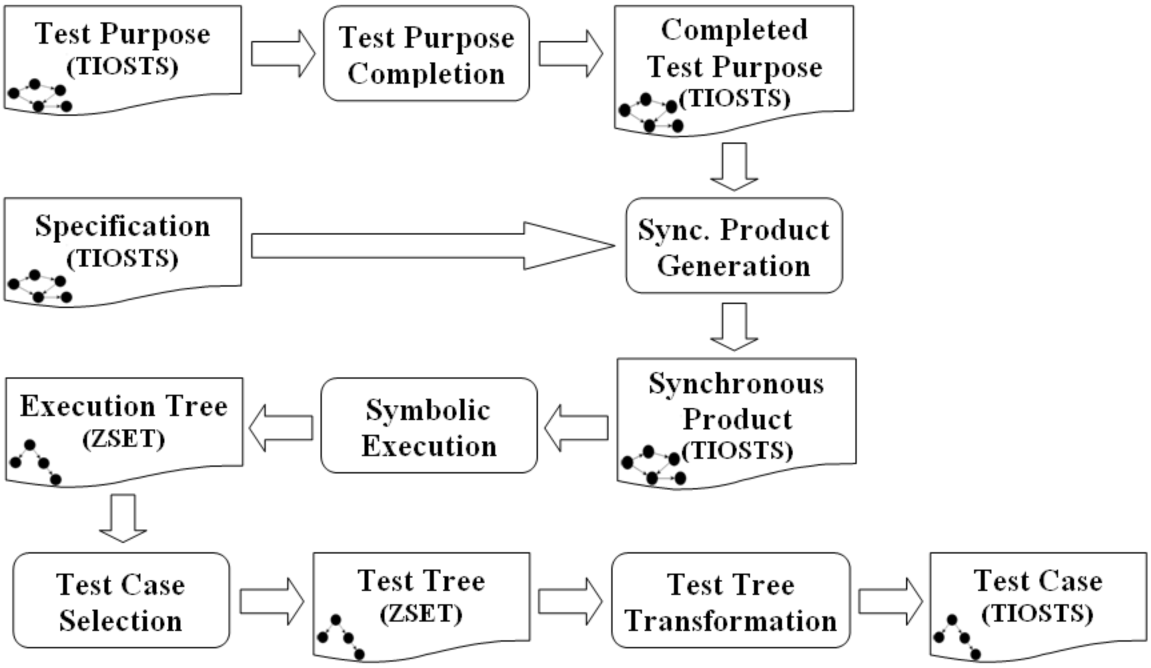


Figura 6.2: Test Case Generation Process

the target location of the transition being processed is the *Reject* location, then a self loop is created with this new transition (Line 24); otherwise, the target location of this new transition is set to the *Reject* location (Line 26). Finally, the created transition is added to the test purpose if it has not been added (Lines from 28 to 30).

Using the asymptotic notation, the running time of Algorithm 6.1 is $O(|L_{TP}| \cdot |\Sigma_{TP}|)$, where $| L_{TP} |$ is the number of locations of the test purpose and $| \Sigma_{TP} |$ is the size of the alphabet of the test purpose.

Figure 6.3 shows the completed test purpose generated by Algorithm 6.1 using, as parameter, the test purpose of Figure 6.1.

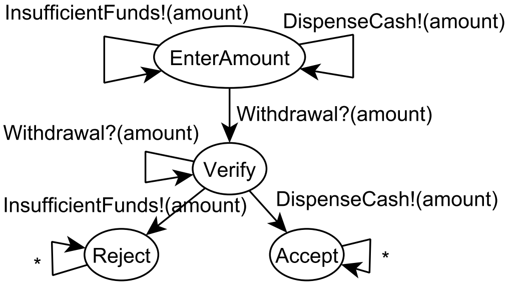


Figura 6.3: Completed Test Purpose Example

Algorithm 6.2: Synchronous Product of W_1 and W_2

```
1 synchronousProduct(TIOSTS $W_1$, TIOSTS $W_2$, TIOSTS syncProduct){
2   if(isCompatible($W_1$, $W_2$)){
3     product($l_1^0$, $l_2^0$, syncProduct);
4     mirror(syncProduct);
5   }
6 }
```

6.2.2 Synchronous Product Generation

Algorithms 6.2 and 6.3 present a simplified implementation of the synchronous product operation defined in Section 5.3. Algorithm 6.2 requires three TIOSTS as parameters: the synchronous product is computed between W_1 (specification) and W_2 (completed test purpose), and the result is returned in *syncProduct*.

Firstly, it is necessary to check whether W_1 and W_2 are compatible for synchronous product (Algorithm 6.2, Line 2) according to Definition 5.3. Then, the *product* method is used to traverse both TIOSTS from their initial locations following the Depth-First Search (DFS) strategy (Algorithm 6.2, Line 3).

Once the synchronous product is computed, the last action of Algorithm 6.2 (Line 4) is to invert input and output actions, in other words, all input actions become output actions and all output actions become input actions. This is important because during the test case execution the inputs of the SUT are outputs of the environment (tester) and vice-versa.

The *product* method is detailed in Algorithm 6.3. Three parameters are needed: l_1, the current location of W_1 being processed; l_2, the current location of W_2 being processed; and *syncProduct*, the synchronous product being computed. The first step (Algorithm 6.3, Line 2) is to check whether l_2 is a verdict location (Accept or Reject location). If so, the processing is stopped.

The loop in Line 5 of Algorithm 6.3 processes each transition t_1 leaving l_1 (i.e., the current location of W_1). $\mathcal{T}_2$ contains all transitions leaving the current location of W_2 that have the same action as t_1 (Algorithm 6.3, Line 6).

When $\mathcal{T}_2$ is empty (Algorithm 6.3, Line 9) means that there is no transition leaving the current location of W_2 with the same action as t_1. In this case, the new transition t of the

Algorithm 6.3: Product of W_1 and W_2

```
product(Location $l_1$, Location $l_2$, TIOSTS syncProduct) {
  if(isVerdict($l_2$)) {
    return;
  }
  for (Transition $t_1$ : $l_1$.getOutGoingTransitions()) {
    Set $\mathcal{T}_2$ := getTransitionsByAction($l_2$, $t_1$.getAction());
    Location source := new Location($l_1$.getLabel() + "_" + $l_2$.getLabel());
    Transition t := new Transition();
    if($\mathcal{T}_2$.isEmpty()){
      t.setSource(source);
      t.setGuard($t_1$.getGuard());
      t.setAction($t_1$.getAction());
      t.setDeadline($t_1$.getDeadline());
      t.setAssignments($t_1$.getAssignments());
      t.setTarget($t_1$.getTarget().getLabel + "_" + $l_2$.getLabel());
      if(!syncProduct.containsTransition(t)) {
        syncProduct.addTransition(t);
        product($t_1$.getTarget(), $l_2$, syncProduct);
      }
    } else {
      for (Transition $t_2$ : $\mathcal{T}_2$) {
        t.setSource(source);
        t.setGuard($t_1$.getGuard() + "AND" + $t_2$.getGuard());
        t.setAction($t_1$.getAction());
        t.setDeadline($t_1$.getDeadline());
        t.setAssignments($t_1$.getAssignments() + $t_2$.getAssignments());
        t.setTarget($t_1$.getTarget().getLabel() + "_" +
                               $t_2$.getTarget().getLabel());
        if(!syncProduct.containsTransition(t)) {
          syncProduct.addTransition(t);
          product($t_1$.getTarget(), $t_2$.getTarget(), syncProduct);
        }
      }
    }
  }
}
```

synchronous product will be identical to t_1 (Lines from 11 to 14) excepting the source (Lines 7 and 10) and target (Line 15) locations. The test of Line 16 is necessary to avoid the inclusion of a transition twice when there are loops in the models and to guarantee that the algorithm terminates. If t has not been added to *syncProduct*, it is added and the algorithm continues recursively with the following parameters: l_1 becomes the target location of t_1, the same l_2 being processed, and *syncProduct*.

If $\mathcal{T}_2$ has one or more transitions with the same action as t_1, each transition $t_2 \in \mathcal{T}_2$ leads to creation of a new transition t to be added to *syncProduct* with the following properties: (1) the source location is the composition of the current locations (Lines 7 and 22); (2) the guard is the conjunction of the guards of t_1 and t_2 (Line 23); (3) the action is the same as t_1 (Line 24); (4) the deadline is the same as t_1 (Line 25); (5) the set of assignments is the union of the assignments set of t_1 and t_2 (Line 26); (6) the target location is the composition of target location of t_1 and target location of t_2 (Line 27).

If t has not been added to *syncProduct*, it is added and the algorithm continues recursively with target locations of t_1 and t_2, and *syncProduct* as parameters (Algorithm 6.3, Lines from 28 to 30).

Algorithms 6.2 and 6.3 have the same running time: $O(\mid \mathcal{T}_1 \mid + \mid \mathcal{T}_2 \mid)$, where $\mid \mathcal{T}_1 \mid$ is the number of transitions of the TIOSTS W_1 and $\mid \mathcal{T}_2 \mid$ is the number of transitions of the TIOSTS W_2.

Figure 6.4 shows the synchronous product generated by Algorithms 6.2 and 6.3 from specification of Figure 5.1 and completed test purpose of Figure 6.3.

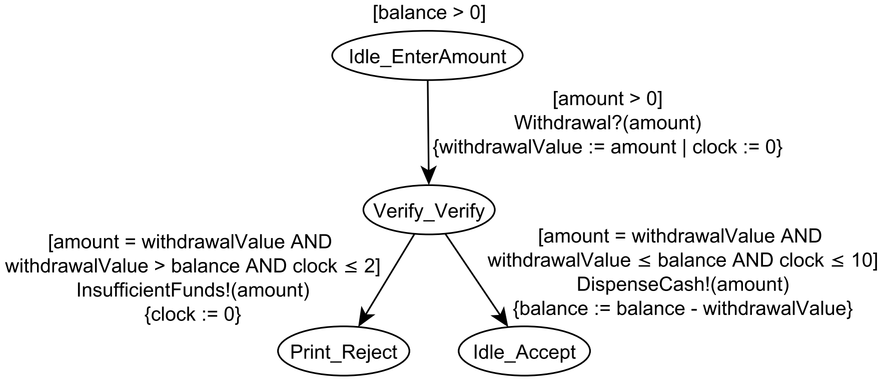

[balance > 0]
Idle_EnterAmount
[amount > 0]
Withdrawal?(amount)
{withdrawalValue := amount | clock := 0}
Verify_Verify
[amount = withdrawalValue AND
withdrawalValue > balance AND clock ≤ 2]
InsufficientFunds!(amount)
{clock := 0}
[amount = withdrawalValue AND
withdrawalValue ≤ balance AND clock ≤ 10]
DispenseCash!(amount)
{balance := balance - withdrawalValue}
Print_Reject
Idle_Accept

Figura 6.4: Synchronous Product Example

6.2.3 Symbolic Execution

Symbolic execution is a technique for analysing programs based on symbolic values as input rather than concrete values [33; 71]. Symbolic execution techniques were used by Gaston *et al.* [53] and Jöbstl *et al.* [66] for test generation for untimed systems. We here extend the work proposed by Jöbstl *et al.* [66] to deal with TIOSTS models.

The main idea is to symbolically execute TIOSTS models using the same technique used for symbolically executing programs. Thus, all possible traces are identified using symbolic values instead of concrete values for action parameters and variables of the model, avoiding the state space explosion problem w.r.t. the data part since data values are not enumerated. The resulting traces are represented as a zone-based symbolic execution tree (Definition 6.6), whose nodes are zone-based symbolic extended states (Definition 6.4) and edges are symbolic actions (Definition 6.5).

Definition 6.4 (Zone-Based Symbolic Extended State)**.** *A zone-based symbolic extended state (ZSES) of a TIOSTS* $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ *is a tuple* $\eta = \langle l, \pi, \varphi, Z \rangle$*, where:*

- $l \in L$ *is a location of* W*;*
- π *is the path condition, that is, a Boolean expression representing a data guard;*
- φ *is a mapping from variables and action parameters to their symbolic values;*
- Z *is a zone representing the solution set of a clock constraint.*

◇

Symbolically executing a TIOSTS implies that data and time must be taken into account. As in [66], path conditions are checked using constraint solving. However, our definition of states differs from [66] because zones are used to check the reachability of states w.r.t. time requirements: a state is reachable if its path condition π is satisfiable and its zone Z is not empty. Zones provide an efficient symbolic representation of time requirements, avoiding the state space explosion problem w.r.t. the time part. Furthermore, ZSESs are connected through transitions labelled by symbolic actions (Definition 6.5).

Definition 6.5 (Symbolic Action)**.** *A symbolic action is a tuple* $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$*, where:*

- *$a \in \Sigma$ is the corresponding action in the TIOSTS;*
- *μ_{sa} is a list of unique identifiers denoting the action parameters of sa;*
- *φ_{sa} is a mapping from the original action parameter names to the unique identifiers in μ_{sa}.*

$\diamond$

We are now ready to define zone-based symbolic execution trees:

Definition 6.6 (Zone-Based Symbolic Execution Tree)**.** *A zone-based symbolic execution tree (ZSET) is a deterministic, connected, acyclic graph represented by a tuple $\langle S, SA, \eta^0, T\rangle$, where:*

- *S is a finite set of zone-based symbolic extended states;*
- *SA is a finite set of symbolic actions;*
- *$\eta^0 \in S$ is the initial zone-based symbolic extended state;*
- *T is a finite set of transitions. Each transition $t \in T$ is a tuple $\langle \eta, sa, \eta' \rangle$, where:*
  - *$\eta \in S$ is the origin state of the transition,*
  - *$sa \in$ SA is the symbolic action of the transition,*
  - *$\eta' \in S$ is is the destination state of the transition.*

$\diamond$

A ZSET is deterministic if $\forall \eta, \eta', \eta'' \in S$, $\forall sa \in$ SA : $\langle \eta, sa, \eta' \rangle \in T \wedge \langle \eta, sa, \eta'' \rangle \in T \Rightarrow \eta' = \eta''$.

Algorithms for symbolically executing symbolic transition systems have been proposed by Gaston et al. [53] and Jöbstl et al. [66]. However, as they do not deal with time, a new algorithm is presented in this thesis.

Algorithm 6.4 is an extended version of the algorithm proposed by Jöbstl et al. [66]. It requires two parameters: TIOSTS W is the input model to be symbolically executed and *ZSET* is the resulting zone-based symbolic execution tree. Firstly, a unique symbolic value is generated for each variable of V and each action parameter of P (Line 2). In Line 3, the

Algorithm 6.4: Symbolic Execution of $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$

```
1  symbolicExecution(TIOSTS W, ZSET ZSET){
2    $\varphi^0 \leftarrow$ map of variables of $V \cup P$ to symbolic values
3    $\eta^0 \leftarrow \langle l^0, \Theta, \varphi^0, Z^0 \rangle$
4    addState(ZSET, $\eta^0$)
5    Unvisited $\leftarrow \{\eta^0\}$
6    while Unvisited $\neq \emptyset$ do
7      pick and remove some $\eta = \langle l, \pi, \varphi, Z \rangle$ from Unvisited
8      for all $\langle l, a, G, A, y, l' \rangle \in \mathcal{T}$ do
9        $\mu_{sa} \leftarrow$ list of unique symbolic values for every parameter of $a$
10       $\varphi_{sa} \leftarrow$ map of action parameters to symbolic values
11       $sa \leftarrow \langle a, \mu_{sa}, \varphi_{sa} \rangle$
12       $\pi' \leftarrow \pi \wedge \varphi(\varphi_{sa}(G^D))$ // $G^D$ is the data guard of $G$
13       $\varphi' \leftarrow \varphi \circ \varphi_{sa} \circ A^D$ // $A^D$ represents data assignments of $A$
14       $Z' \leftarrow [A^C \leftarrow 0](G^C \cap \vec{Z})$ // $A^C$ is the set of clocks to reset and $G^C$ is the clock guard of $G$
15       $\eta' = \langle l', \pi', \varphi', Z' \rangle$
16       if (isReachable($\eta'$) $\wedge \neg$(upperBoundReached($l'$)) $\wedge\ \eta' \not\subseteq \eta''\ \forall \eta'' \in$ ZSET) then
17           Unvisited $\leftarrow$ Unvisited $\cup \{\eta'\}$
18           addState(ZSET, $\eta'$)
19           addTransition(ZSET, $\langle \eta, sa, \eta' \rangle$)
20       end if
21     end for
22   end while
23 }
```

first state η^0 of *ZSET* is defined considering the initial location of W, the initial condition of W as first path condition, the mapping defined in Line 2, and the initial clock zone (i.e., all clocks set to zero). Once defined, the first state η^0 is added to *ZSET* (Line 4).

The state η^0 is added to the set of states to be visited (Line 5). As long as there are unvisited states (Line 6), the algorithm picks and remove some state η from *Unvisited* (Line 7). The ZSES η refers to a location l of W and the loop in Line 8 processes all transitions from l.

The symbolic action sa is computed from the action a, attributing unique symbolic values for every parameter of a (Line 9) and mapping the original action parameter names to the defined symbolic values (Line 10).

Once the symbolic action has been defined (Line 11), the next step of Algorithm 6.4 is to compute the target state η'. Thus, the path condition π' for η' is defined (Line 12) as the conjunction of π with the guard G^D (i.e., the data guard of G) considering the mappings φ and φ_{sa}. The mapping φ' is defined through $\varphi \circ \varphi_{sa} \circ A^D$ (Line 13), where A^D represents data assignments of A and $\circ$ denotes function composition.

Z' is defined in Line 14. The successor of Z is defined by letting time elapse ($\vec{Z}$), taking the intersection with the clock guard G^C, and finally updating the values of clocks that are reset (i.e., clocks in A^C).

Once π', φ', and Z' have been defined, the target state η' is created in Line 15. Finally, η' is added to the set of states to be visited (Line 17) and a new transition labelled by sa connecting η to η' is added to *ZSET* (Lines 18 and 19), if the following conditions are satisfied (Line 16):

1. The state η' is reachable, that is, the path condition π' is satisfiable and the zone Z' is not empty;

2. The number of ZSESs in the current path that correspond to the location l' does not exceed a certain bound. This checking is needed to avoid infinite ZSETs in the case where there are loops in the specification whose number of iterations depends on values assigned to parameters and variables [66];

3. $\eta' \not\subseteq \eta'' \;\; \forall \eta'' \in$ *ZSET* according to Definition 6.7, where the state inclusion of Gaston et al. [53] was extended to deal with zones.

Definition 6.7 (ZSESs Comparison). *Let $\eta = \langle l, \pi, \varphi, Z\rangle$ and $\eta' = \langle l', \pi', \varphi', Z'\rangle$ be two zone-based symbolic extended states. ZSES η' is included in ZSES η, that is, $\eta' \subseteq \eta$, if and only if:*

1. $l' = l$;

2. $(\pi' \wedge \bigwedge_{x \in A^D}(x = \varphi'(x))) \Rightarrow (\pi \wedge \bigwedge_{x \in A^D}(x = \varphi(x)))$ is a tautology, where A^D represents data assignments of the TIOSTS;

3. $Z' \subseteq Z$.

◇

As the implementation of the proposed algorithm for symbolic execution depends on tools related to concepts that are outside the scope of this thesis such as zones and constraint solving, the running time of the algorithm is described independently of implementation strategies and tools. Thus, the running time of Algorithm 6.4 is $O(\mid L \mid + \text{Cost(reachability analysis)} + \text{Cost(ZSESs comparison)})$, where $\mid L \mid$ is the number of locations of the TIOSTS W, Cost(reachability analysis) is the cost to verify whether a ZSES is reachable, and Cost(ZSESs comparison)) is the cost to compare two ZSESs.

Figure 6.5 presents the ZSET obtained from the symbolic execution of the synchronous product shown in Figure 6.4.

6.2.4 Test Case Selection

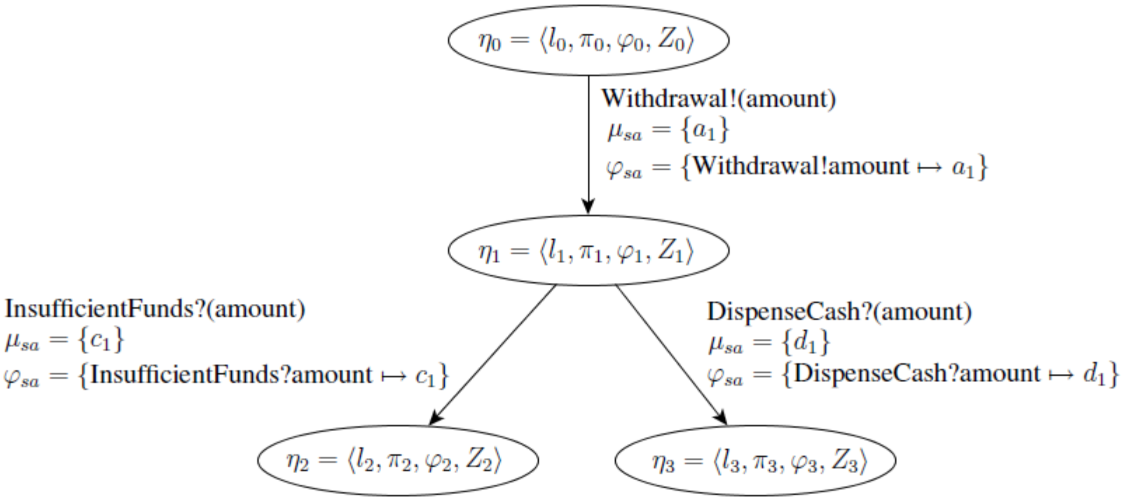
Once all possible traces have been identified by symbolic execution, the next step is to select a test case by choosing a trace that leads to an *Accept* state. For this, it is necessary to select a subtree of the generated ZSET called test tree. Finally, the selected test tree is translated into a test case (see Subsection 6.2.5).

The strategy used for the selection of the test tree is the same proposed by Jöbstl *et al.* [66], which is similar to the strategy of the TGV tool [62]. The idea is to select one reachable *Accept* state and perform a backward traversal to the root ZSES. Finally, a forward traversal is performed in order to extend the selected path to a test tree by adding missing inputs that are allowed by the specification. These missing inputs are possible outputs of the SUT and they are important to avoid fail verdicts on outputs allowed by the specification. In this case, the verdict is **Inconclusive**. Note that the forward traversal ensures the controllability of the generated test tree (i.e. test cases do not have the choice between inputs and outputs, or between several outputs).

The test tree from the ZSET in Figure 6.5 is the same ZSET since there is only one path leading to an *Accept* state and the addition of missing inputs leads to the whole ZSET.

6.2.5 Test Tree Transformation

Considering the conformance testing framework defined in Section 6.1, test cases are timed input-output symbolic transition systems. Thus, the last step of the test case generation


$\eta_0 = \langle l_0, \pi_0, \varphi_0, Z_0\rangle$
Withdrawal!(amount)
$\mu_{sa} = \{a_1\}$
$\varphi_{sa} = \{\text{Withdrawal!amount} \mapsto a_1\}$
$\eta_1 = \langle l_1, \pi_1, \varphi_1, Z_1\rangle$
InsufficientFunds?(amount)
$\mu_{sa} = \{c_1\}$
$\varphi_{sa} = \{\text{InsufficientFunds?amount} \mapsto c_1\}$
DispenseCash?(amount)
$\mu_{sa} = \{d_1\}$
$\varphi_{sa} = \{\text{DispenseCash?amount} \mapsto d_1\}$
$\eta_2 = \langle l_2, \pi_2, \varphi_2, Z_2\rangle$
$\eta_3 = \langle l_3, \pi_3, \varphi_3, Z_3\rangle$

$l_0 = \text{Idle\_EnterAmount}$
$\pi_0 = b_0 > 0$
$\varphi_0 = \{\text{balance} \mapsto b_0, \text{withdrawalValue} \mapsto w_0, \text{Withdrawal!amount} \mapsto a_0,$
$\text{InsufficientFunds?amount} \mapsto c_0, \text{DispenseCash?amount} \mapsto d_0\}$
$Z_0 = \text{clock} = 0$

$l_1 = \text{Verify\_Verify}$
$\pi_1 = b_0 > 0 \text{ AND } a_1 > 0$
$\varphi_1 = \{\text{balance} \mapsto b_0, \text{withdrawalValue} \mapsto a_1, \text{Withdrawal!amount} \mapsto a_1,$
$\text{InsufficientFunds?amount} \mapsto c_0, \text{DispenseCash?amount} \mapsto d_0\}$
$Z_1 = \text{clock} = 0$

$l_2 = \text{Print\_Reject}$
$\pi_2 = b_0 > 0 \text{ AND } a_1 > 0 \text{ AND } c_1 = a_1 \text{ AND } a_1 > b_0$
$\varphi_2 = \{\text{balance} \mapsto b_0, \text{withdrawalValue} \mapsto a_1, \text{Withdrawal!amount} \mapsto a_1,$
$\text{InsufficientFunds?amount} \mapsto c_1, \text{DispenseCash?amount} \mapsto d_0\}$
$Z_2 = \text{clock} \leq 2$

$l_3 = \text{Idle\_Accept}$
$\pi_3 = b_0 > 0 \text{ AND } a_1 > 0 \text{ AND } d_1 = a_1 \text{ AND } a_1 \leq b_0$
$\varphi_3 = \{\text{balance} \mapsto b_0 - a_1, \text{withdrawalValue} \mapsto a_1, \text{Withdrawal!amount} \mapsto a_1,$
$\text{InsufficientFunds?amount} \mapsto c_0, \text{DispenseCash?amount} \mapsto d_1\}$
$Z_3 = \text{clock} \leq 10$

Figura 6.5: Zone-Based Symbolic Execution Tree of the TIOSTS of Figure 6.4

process (Figure 6.2) is to translate the selected test tree *TT* $= \langle S, \text{SA}, \eta^0, T\rangle$ into a test case TIOSTS *TC* $= \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$.

The test tree translation operation is described by Algorithm 6.5. It requires three parameters: TIOSTS *SP* is the synchronous product from which the test tree was obtained, ZSET *ZSET* is the test tree, and TIOSTS *TC* is the resulting test case.

The data of *TC* (i.e. $V \cup P$) is defined by symbolic values of *ZSET* (Lines 2 and 3). As in [66], the symbolic values are considered as variables and parameters of the test case. Let $\eta^0 = \langle l^0, \pi^0, \varphi^0, Z^0\rangle$ be the initial state of *ZSET*, then the initial condition of *TC* is π^0

Algorithm 6.5: Test Tree Translation Algorithm

```
1  ZSET2TC(TIOSTS $SP$, ZSET $ZSET$, TIOSTS $TC$){
2     $V_{TC} \leftarrow$ symbolic values of variables of ZSET
3     $P_{TC} \leftarrow$ symbolic values of parameters of ZSET
4     // Let $\eta^0 = \langle l^0, \pi^0, \varphi^0, Z^0 \rangle$ be the initial state of ZSET
5     $\Theta_{TC} \leftarrow \pi^0$
6     $L_{TC} \leftarrow S$
7     $l^0_{TC} \leftarrow \eta^0$
8     $\Sigma_{TC} \leftarrow \bigcup_{\langle a,\mu_{sa},\varphi_{sa} \rangle \in SA} a$
9     $C_{TC} \leftarrow C_{SP}$
10    for all $\langle \eta, sa, \eta' \rangle \in T_{ZSET}$ do
11       $l \leftarrow \eta$
12       $a \leftarrow$ action of $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$ with parameters of $\mu_{sa}$
13       $G \leftarrow$ conjunction of path condition of $\eta'$ with clock guards associated with $a$ in SP
14       $A \leftarrow$ clock resets associated with $a$ in SP
15       $y \leftarrow$ deadline associated with $a$ in SP
16       $l' \leftarrow \eta'$
17       addTransition$(TC, \langle l, a, G, A, y, l' \rangle)$
18    end for
19 }
```

(Line 5), the set of locations is S (Line 6), the initial location is η^0 (Line 7), the alphabet is $\bigcup_{\langle a,\mu_{sa},\varphi_{sa} \rangle \in SA} a$ (Line 8), and the set of clocks is the same as the synchronous product *SP*, that is, C_{SP} (Line 9).

All transitions $\langle \eta, sa, \eta' \rangle \in T$ of *ZSET* are analysed in Lines from 10 to 18. Each transition of *ZSET* leads to the creation of a new transition $\langle l, a, G, A, y, l' \rangle \in \mathcal{T}$ in the test case. Thus, the source location is η, the action of the new transition is the action of $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$ with parameters of μ_{sa}, the conjunction of the path condition of η' with clock guards associated with a in *SP* is the guard, the assignments are defined based on clock resets associated with a in *SP*, the deadline is the same as the one associated with a in *SP*, and the target location is η'.

Using the asymptotic notation, the running time of Algorithm 6.5 is $O(|T|)$, where $|T|$ is the number of transitions of the *ZSET* test tree.

Figure 6.6 presents the test case obtained from the ZSET of Figure 6.5. It starts by performing a withdrawal transaction in the ATM system and resetting the clock to zero. Then it expects to receive the money. If the ATM system dispenses the expected money in at most 10 time units, the verdict is **Pass**, that is, the implementation is in conformance with the specification and the test purpose. If the ATM system indicates insufficient funds in at most 2 time units, the verdict is **Inconclusive** (i.e. the implementation conforms to the specification but the desired behaviour was not observed). Finally, if either an unspecified output is received or a time requirement is not satisfied, the verdict is **Fail**.

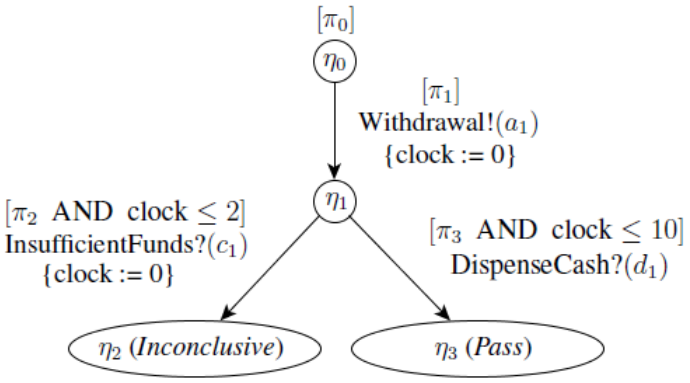


Figura 6.6: Test Case Obtained from the ZSET of Figure 6.5

6.3 Properties of the Test Cases

This section comments on properties of the test cases generated by the process presented in Section 6.2. The execution of test cases must be formalised in order to establish some properties such as soundness and exhaustiveness, where the conformance relation is linked to the verdicts.

The generated test cases are considered as a mechanism for guiding the execution of the implementation. Thus, conformance checking is performed in two steps, in an offline way. Firstly, the implementation is executed, guided by test cases, and all information needed to check conformance (e.g., input actions, responses, and time associated with responses) are logged into a file. Considering that the SUT runs on a real-time environment such as a real-time operating system, it is important that the implementation logs its own information in order to reduce the number of processes and consequently avoid introduction of noise in the results.

As said, each SUT execution produces a log describing the exercised scenario. This log is an *observable trace* (defined in Subsection 5.2), which is a specific sequence of observable discrete and time-elapsing actions. For example, considering the TIOSTS of Figure 5.1 an observable trace of a scenario where a withdrawal transaction is successfully done in 5 time units could be represented by $\sigma_{\text{SUT}} = 0\ \mathit{Withdrawal?}(100)\ 5\ \mathit{DispenseCash!}(100)$.

Let $[\![TC]\!] = \langle S, S^0, Act, T\rangle$ be the TIOLTS semantics of the test case $TC = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$. Thus, an observable trace of $\mathcal{I}$ can be checked with respect to the test case through the TIOLTS parallel composition defined by Krichen and Tripakis [78]. In this case, each trace $\sigma \in \mathit{Traces}([\![TC]\!] \mid\mid \mathit{ObservableTraces}(\mathcal{I}))$ is associated with one of the following scenarios:

- If all outputs of TC are executed and all inputs are observed on time, then the resulting verdict is **Pass**, that is, $\mathit{verdict}(\sigma) = \textbf{Pass} \triangleq S^0\ \mathit{after}\ \sigma \subseteq \mathit{Pass}$;

- If, at any moment, any unspecified input is observed by the test case or some time requirement is not met, the conformance checking is stopped and the resulting verdict is **Fail**, that is, $\mathit{verdict}(\sigma) = \textbf{Fail} \triangleq S^0\ \mathit{after}\ \sigma \subseteq \mathit{Fail}$;

- We denote $\mathit{verdict}(\sigma) = \textbf{Inconclusive} \triangleq S^0\ \mathit{after}\ \sigma \subseteq \mathit{Inconclusive}$ for two situations: if $\mathcal{I}$, at any moment, blocks or spends a lot of time to emit an output; and if the outputs of $\mathcal{I}$ are specified by $\mathcal{S}$ but the behaviour specified by a test purpose is not exhibited.

Given the possible situations with their respective verdicts, the rejection of $\mathcal{I}$ by a test case TC is formally defined as follows:

Definition 6.8 (may reject)**.** TC may reject $\mathcal{I} \triangleq \exists\sigma \in \text{Traces}([\![TC]\!] \mid\mid \text{ObservableTraces}(\mathcal{I}))$: verdict*(σ) =* ***Fail****.* ◇

The conformance relation of an implementation with its specification is decided based on verdicts obtained with the execution of test cases. So, Definition 6.9 formally relates the **tioco** relation to the verdicts considering some properties of test cases and test suites.

Definition 6.9 (Soundness and Exhaustiveness)**.** *A test case TC is* sound *for $\mathcal{S}$ and* ***tioco*** *if $\forall\mathcal{I}$, $\mathcal{I}$* ***tioco*** *$\mathcal{S} \Rightarrow \neg($$TC$ may reject $\mathcal{I}$). A test suite is* sound *if all its test cases are* sound *and it is* exhaustive *for $\mathcal{S}$ and* ***tioco*** *if $\forall\mathcal{I}$, $\neg$($\mathcal{I}$* ***tioco*** *$\mathcal{S}$) $\Rightarrow \exists TC : TC$* may reject $\mathcal{I}$*. Finally, a test suite is* complete *if it is both* sound *and* exhaustive. ◇

Informally, a test suite is sound if correct implementations are never rejected. On the other hand, a test suite is exhaustive if all non-conforming implementations are rejected. A test suite that can identify all conforming and non-conforming implementations is called complete. Since a complete test suite is a very strong requirement for practical testing, sound test suites are more commonly accepted. In this context, the test cases generated by our approach have the properties stated in Theorem 6.1.

Theorem 6.1. *For every specification $\mathcal{S}$, all test suites generated by our approach are sound. Moreover, test suites can be considered as being exhaustive when they refer to specific scenarios defined by test purposes.* $\diamond$

The proof of Theorem 6.1 is not detailed here but the main ideas are discussed (see detailed proofs in Appendix A). For soundness, we need to prove that if a test case TC may reject $\mathcal{I}$ (implementing the specification $\mathcal{S}$), then $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$. In this case, we only need to prove that a **Fail** verdict only occurs if $\mathcal{I}$ emits an unspecified output or some time requirement is not met. In our approach, test cases are generated based on symbolic execution of specifications. This approach allows to identify all possible traces of a specification. Thus, the unique case where a **Fail** verdict occurs is exactly when $\mathcal{I}$ emits an unexpected output or some time requirements is not satisfied. For exhaustiveness, we need to prove that for every non-conforming $\mathcal{I}$ there is a test purpose TP and a way of generating a test case TC from $\mathcal{S}$ and TP, such that TC may reject $\mathcal{I}$. Given that $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$, then there is a trace σ of $\mathcal{S}$ such that an output of $\mathcal{I}$ after σ is not allowed by $\mathcal{S}$. In this case, a TP can be defined based on σ and used to generate test cases where $\mathcal{I}$ may be rejected.

6.4 Concluding Remarks

This chapter presented an approach to conformance testing of real-time systems based on the use of a symbolic model that abstracts both time and data in order to broadening the application of conformance testing in this field. It also described the test case generation process and discussed some properties of the generated test cases.

The presented test case generation process is completely automated by a tool developed to support the proposed approach. In order to check the satisfiability of path conditions and verify state inclusion w.r.t. data we are using the CVC3 SMT Solver[2]. As the satisfiability of data guards is assumed to be decidable, the CVC3 SMT Solver arises as a promising tool [111]. Moreover, data guards of a TIOSTS are expressed using the same notation as the notation used by the CVC3 SMT Solver, which facilitates the use of the tool and does not require any translation. However, it is important to note that our approach is limited to the types supported by this solver such as Boolean, integer, real, arrays, records, etc.

All operations related to zones used in Algorithm 6.4 are provided by UPPAAL DBM Library[3]. The same library is also used to verify the state inclusion w.r.t. zones.

[2] http://www.cs.nyu.edu/acsys/cvc3

[3] http://www.cs.aau.dk/˜adavid/UDBM

Capítulo 7

Teste de Interrupção em Sistemas de Tempo Real

Este capítulo descreve como modelar e testar interrupções utilizando o arcabouço de teste de conformidade baseado em modelos TIOSTS apresentado nos Capítulos 5 e 6. A estratégia de teste de interrupção definida é somente uma forma de modelagem utilizando modelos TIOSTS e, dessa forma, nenhuma modificação na teoria é necessária. Esta estratégia é mesma apresentada em [7], onde interrupções são modeladas através de modelos simbólicos para sistemas sem requisitos de tempo. Além disso, assim como a estratégia baseada em modelos ALTS apresentada no Capítulo 3, a estratégia de teste de interrupção baseada em modelos TIOSTS possibilita a combinação de interrupções em diferentes pontos e permite a seleção de casos de teste baseada em propósitos de teste.

7.1 Modelling and Testing Interruptions in Real-Time Systems

The TIOSTS model proposed in Chapter 5 can be used to model interruptions. The idea is to take advantage of the use of variables and action parameters in order to guarantee that once the main flow has been interrupted, it can continue its execution from the same point where the interruption had started.

In order to model interruptions, consider the existence of two models: one TIOSTS representing the main flow (Figure 7.1, locations from 0 to 20), that is, the application that can be interrupted; and another TIOSTS representing the interruption (Figure 7.1, locations from 21 to 28). Thus, the main flow can be linked with the interruption model through the following steps:

1. Identify the point (location) where the interruption can occur (Figure 7.1, location 10);

2. Link this point to the interruption behaviour using a transition labelled as follows: the guard is *intCode = X and choice = 0*, where *X* is an integer that uniquely identifies this point of interruption; the action is *Interrupt?(intCode)*; and the assignment is *choice := intCode* (Figure 7.1, transition from location 10 to 21). Notice that the value of the parameter *intCode* is saved into the *choice* variable.

3. Connect the last action of the interruption behaviour to the same point where the interruption started using a transition labelled with the guard *choice = X*, where *X* is the same value that uniquely identifies this point of interruption (Figure 7.1, transition from location 28 to 10). This guard is used to guarantee that the main flow continues its execution from the same point where it had been interrupted. For instance, if an interruption begins with the parameter *intCode* equals to 1, then it must finish performing the action that has the following guard: *choice = 1*.

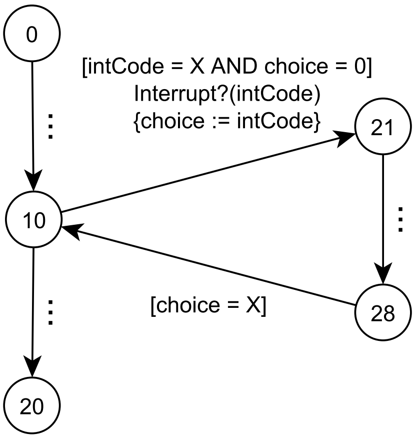


Figura 7.1: Modelling an Abstract Interruption

The test case generation strategy where only one interruption is allowed for each test case is achieved because of the second part of the guard (*choice = 0*) associated to the *Interrupt* action. When an interruption is allowed, the value of the *choice* variable is changed to any value different from zero, then all other interruptions are automatically discarded during the test case generation.

The defined steps must be performed for all points where interruptions can occur. As a complete model with all possibilities of interruption represents many scenarios, the test selection strategy based on test purposes defined in Chapter 6 is used for testing specific interruptions in specific scenarios. For this, it is enough to use the *Interrupt* action in the test purpose carrying the integer that identifies the selected point. For generating test cases without interruptions, the *Interrupt* action is taken to the *Reject* location.

In order to test interruptions using TIOSTS models, the same test architecture presented in Chapter 3 is adopted. Thus, the environment is assumed to be fully controllable by the tester.

7.2 Instantiating the Strategy with an Example

A real-time version of the mobile phone application described in Chapter 3 is used for describing how to deal with interruptions using the TIOSTS formalism. Now, the action of removing a message from inbox after selecting the “Remove” option must be performed in at most 2000 milliseconds. Also, only unblocked messages can be removed and the main application can be interrupted at some points by the Incoming Alert interruption. As said in Chapter 3, this interruption specifies the arrival of a simple text message displayed inside a dialog box. Figure 7.2 shows the TIOSTS model that represents the described behaviour, where locations from 1 to 12 represent the behaviour of removing a message from inbox and locations from 13 to 16 represent the occurrence of interruptions.

As we can see, in Figure 7.2, the interruption model is connected to the feature that can be interrupted (the main flow) using the *Interrupt* action carrying a parameter (*intCode*) that identifies the place where the interruption is allowed. Then, the *intCode* parameter is saved into the *choice* variable. Each point where an interruption is allowed has a different integer value associated with it. Another important information is in the last action of the interruption, where there is a guard used to guarantee the return to the correct point of the main flow.

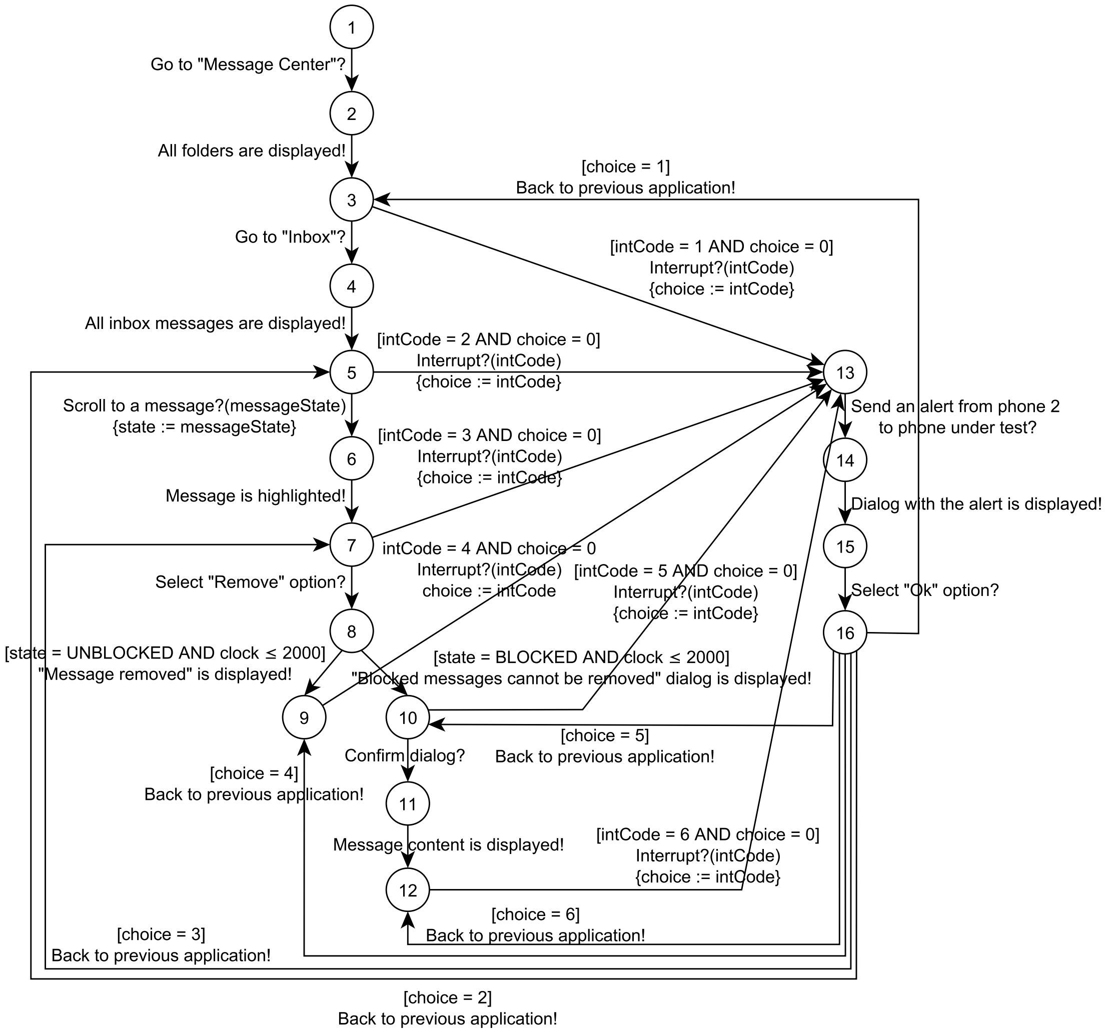
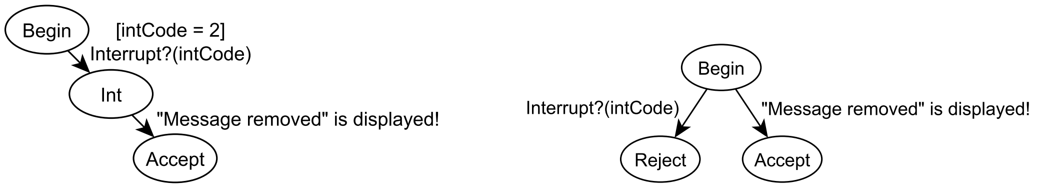

1
Go to "Message Center"?
2
All folders are displayed!
3
[choice = 1]
Back to previous application!
Go to "Inbox"?
[intCode = 1 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
4
All inbox messages are displayed!
[intCode = 2 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
5
13
Scroll to a message?(messageState)
{state := messageState}
Send an alert from phone 2
to phone under test?
[intCode = 3 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
6
14
Message is highlighted!
Dialog with the alert is displayed!
7
15
intCode = 4 AND choice = 0
Interrupt?(intCode)
choice := intCode
[intCode = 5 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
Select "Remove" option?
Select "Ok" option?
8
16
[state = UNBLOCKED AND clock ≤ 2000]
"Message removed" is displayed!
[state = BLOCKED AND clock ≤ 2000]
"Blocked messages cannot be removed" dialog is displayed!
9
10
[choice = 5]
Back to previous application!
Confirm dialog?
[choice = 4]
Back to previous application!
11
Message content is displayed!
[intCode = 6 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
12
[choice = 6]
Back to previous application!
[choice = 3]
Back to previous application!
[choice = 2]
Back to previous application!

Figura 7.2: Real-Time Version of the Remove Message Behaviour with Interruptions

As discussed in Chapter 3, an interruption can occur at infinite points during the system execution, but in the tester's point of view, an interruption can only be observed after an output of the SUT. Thus, the TIOSTS model of Figure 7.2 represents all possibilities of interruption from the tester's point of view.

Once the system is specified using the TIOSTS formalism, the next step is to define test purposes in order to check specific interruptions at some points. Considering the specification in Figure 7.2, a test purpose can be defined in order to verify the scenario where an alert appears when the user is accessing the inbox folder. As the selected interruption point corresponds to the second output of the specification, the action *Interrupt* must carry the integer 2. Next, the last action of the selected behaviour (the scenario where the message is removed) is appended to the test purpose. Figure 7.3(a) presents the test purpose defined above and



(a) Scenario with Interruption (b) Scenario without Interruption

Figura 7.3: Test Purposes

Figure 7.4 shows the obtained test case. Note that in the generated test case (Figure 7.4) the interruption occurs only in one specific point.

Another test purpose can be defined in order to test a scenario where a message is removed and all interruptions are not allowed. This test purpose is presented in Figure 7.3(b). Note that to prohibit all interruptions it is enough to take the *Interrupt* action to the *Reject* location. The other action of the test purpose (*"Message removed" is displayed!*) is used to select the scenario where the message is removed. The generated test case is shown in Figure 7.5.

7.3 Concluding Remarks

This chapter presented a strategy developed for modelling and testing interruptions that is based on the symbolic model proposed in Chapter 5. The presented strategy makes it possible for interruptions to be combined at different points of possibly different flows of execution. Moreover, test purposes can be used to select specific interruptions to be tested. Finally, it is important to remark that this proposed strategy is only a specific way of modelling using the TIOSTS notation. No modifications in the theory and algorithms are needed.

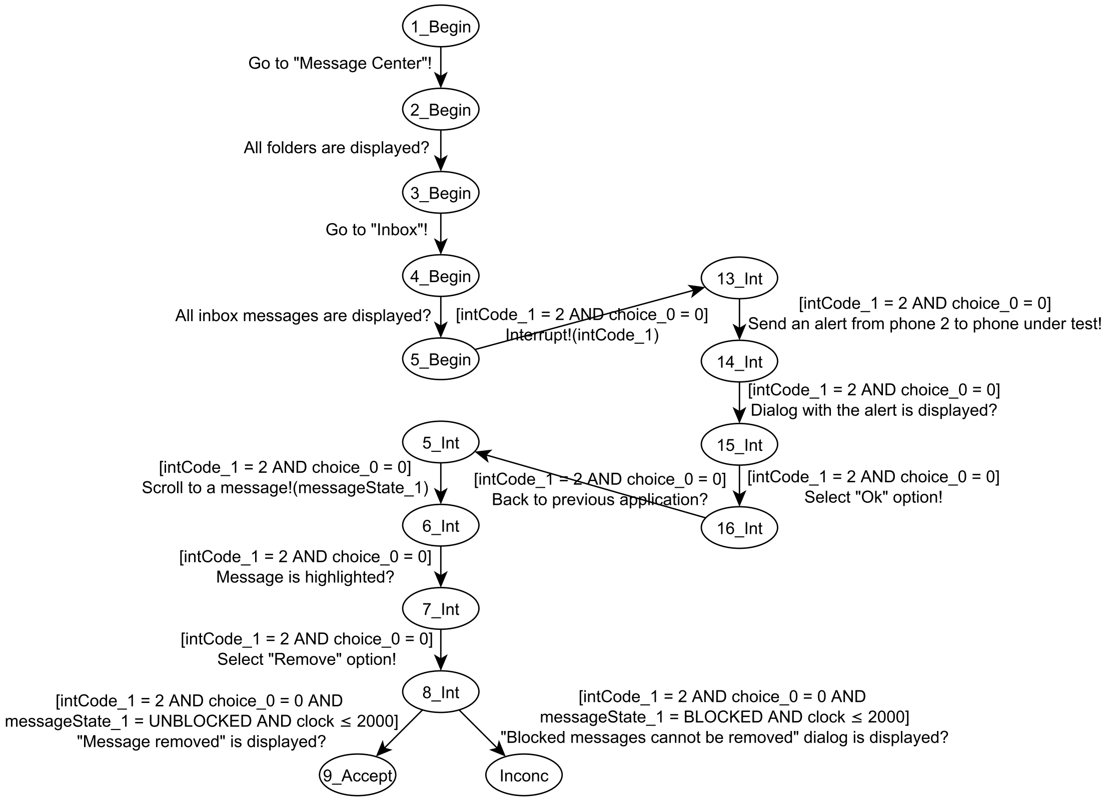

1_Begin
Go to "Message Center"!
2_Begin
All folders are displayed?
3_Begin
Go to "Inbox"!
4_Begin
All inbox messages are displayed?
5_Begin
[intCode_1 = 2 AND choice_0 = 0]
Interrupt!(intCode_1)
13_Int
[intCode_1 = 2 AND choice_0 = 0]
Send an alert from phone 2 to phone under test!
14_Int
[intCode_1 = 2 AND choice_0 = 0]
Dialog with the alert is displayed?
15_Int
[intCode_1 = 2 AND choice_0 = 0]
Select "Ok" option!
16_Int
[intCode_1 = 2 AND choice_0 = 0]
Back to previous application?
5_Int
[intCode_1 = 2 AND choice_0 = 0]
Scroll to a message!(messageState_1)
6_Int
[intCode_1 = 2 AND choice_0 = 0]
Message is highlighted?
7_Int
[intCode_1 = 2 AND choice_0 = 0]
Select "Remove" option!
8_Int
[intCode_1 = 2 AND choice_0 = 0 AND
messageState_1 = UNBLOCKED AND clock ≤ 2000]
"Message removed" is displayed?
[intCode_1 = 2 AND choice_0 = 0 AND
messageState_1 = BLOCKED AND clock ≤ 2000]
"Blocked messages cannot be removed" dialog is displayed?
9_Accept
Inconc

Figura 7.4: Test Case with Interruption

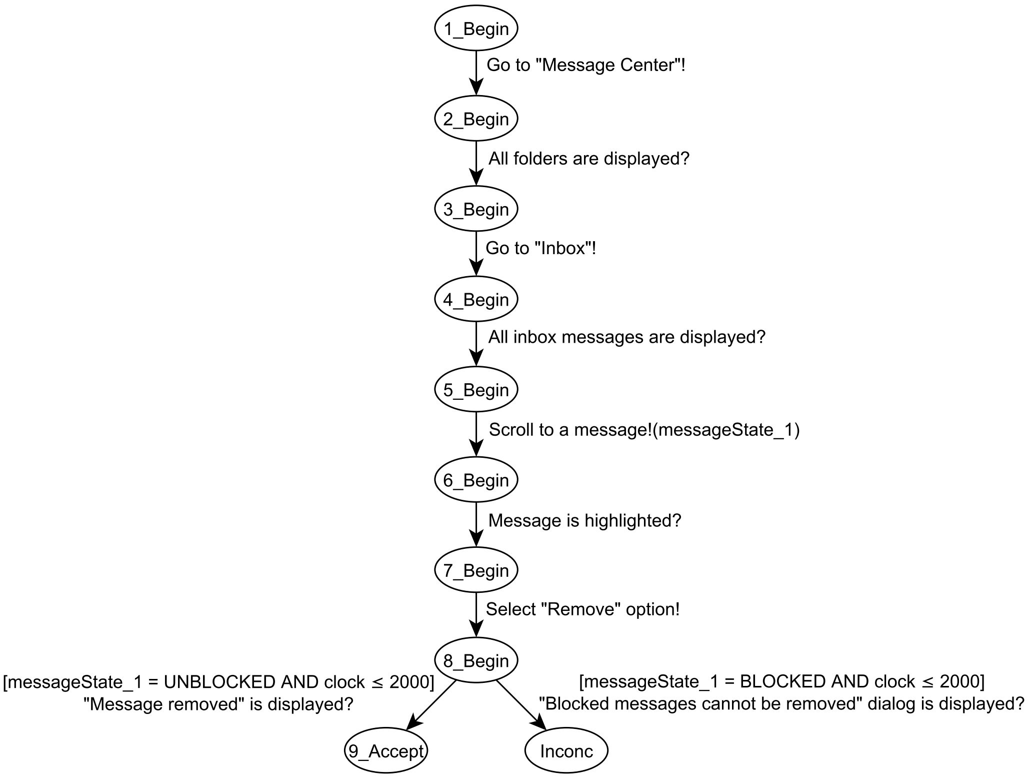

1_Begin
Go to "Message Center"!
2_Begin
All folders are displayed?
3_Begin
Go to "Inbox"!
4_Begin
All inbox messages are displayed?
5_Begin
Scroll to a message!(messageState_1)
6_Begin
Message is highlighted?
7_Begin
Select "Remove" option!
8_Begin
[messageState_1 = UNBLOCKED AND clock ≤ 2000]
"Message removed" is displayed?
[messageState_1 = BLOCKED AND clock ≤ 2000]
"Blocked messages cannot be removed" dialog is displayed?
9_Accept
Inconc

Figura 7.5: Test Case without Interruption

Capítulo 8

Estudos de Caso

O objetivo deste capítulo é apresentar alguns estudos de caso realizados com a finalidade de avaliar a aplicação prática da abordagem baseada em modelos simbólicos proposta nesta tese. A aplicação da abordagem desenvolvida é avaliada utilizando dois estudos de caso: um sistema de alarme contra intrusos e um sistema para veículos guiados automaticamente.

8.1 The Burglar Alarm System

The first case study is aimed at generating and executing test cases for the burglar alarm system described in Section 4.3. For this, a simplified implementation was developed to run on a real-time operating system named FreeRTOS [112], a mini-kernel that can be used to develop real-time systems for embedded devices. The alarm system case study is useful because it is possible to execute test cases and it allows us to show how scenarios with interruptions can be checked.

The main objective of this case study is to assess the performance of the symbolic model-based approach developed for testing real-time systems. In order to achieve this objective we use the Goal/Question/Metric (GQM) paradigm [14], a mechanism for defining and evaluating goals using measurement.

8.1.1 The GQM Measurement Model

The GQM paradigm is a top-down systematic approach to evaluating goals based on an operational level. Thus, the first step is to define the goal to be evaluated (conceptual level). Secondly, at the operational level, questions are defined in order to characterize the measurement object with respect to desired quality criteria. Finally, a set of data is defined to answer each question in a quantitative level.

Figure 8.1 presents the GQM measurement model defined for this case study. The main goal is to evaluate the performance of the symbolic model-based approach developed for testing real-time systems. Thus, three questions were defined to characterize the measurement objects:

What is the effort required to use this approach? This question is intended to evaluate the effort required to apply all the test process from the building of the model to test case execution. For answering this question the following metric was defined: $E = E_1 + E_2 + E_3 + E_4 + E_5$, where:

- E_1 is the time spent to build the model using the TIOSTS formalism;
- E_2 is the time spent to define test purposes in order to test specific scenarios. Test purposes are also defined using the TIOSTS formalism;
- E_3 is the time spent to generate test cases using the prototype tool implementing the presented algorithms;
- E_4 is the time spent to implement the automatically generated test cases;
- E_5 is the time spent to execute the implemented test cases and evaluate the obtained results for emitting verdicts.

How effective is this approach? This question is intended to evaluate the effectiveness of the proposed approach w.r.t. fault coverage. The C metric, used to answer this question, indicates the ability of generated test cases of uncovering faults described by a previously defined fault model.

What percentage of test cases cannot be executed? This question is intended to identify the percentage of invalid test cases, represented by the metric I.

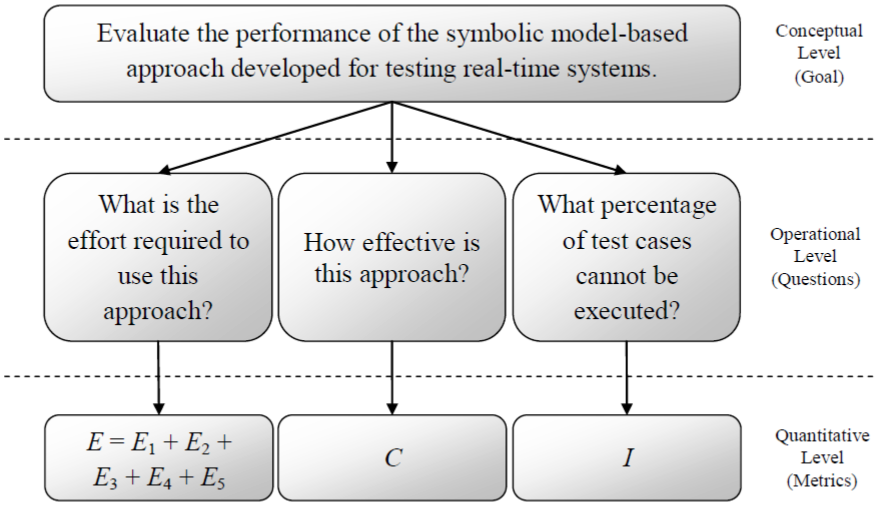

Evaluate the performance of the symbolic model-based approach developed for testing real-time systems.
Conceptual Level (Goal)
What is the effort required to use this approach?
How effective is this approach?
What percentage of test cases cannot be executed?
Operational Level (Questions)
$E = E_1 + E_2 + E_3 + E_4 + E_5$
C
I
Quantitative Level (Metrics)

Figura 8.1: Measurement Model for the Alarm System Case Study

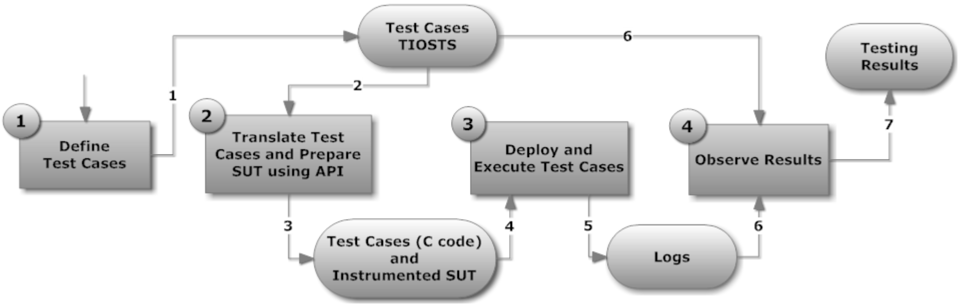

Test Cases TIOSTS
Testing Results
1 Define Test Cases
2 Translate Test Cases and Prepare SUT using API
3 Deploy and Execute Test Cases
4 Observe Results
Test Cases (C code) and Instrumented SUT
Logs
1
2
3
4
5
6
6
7

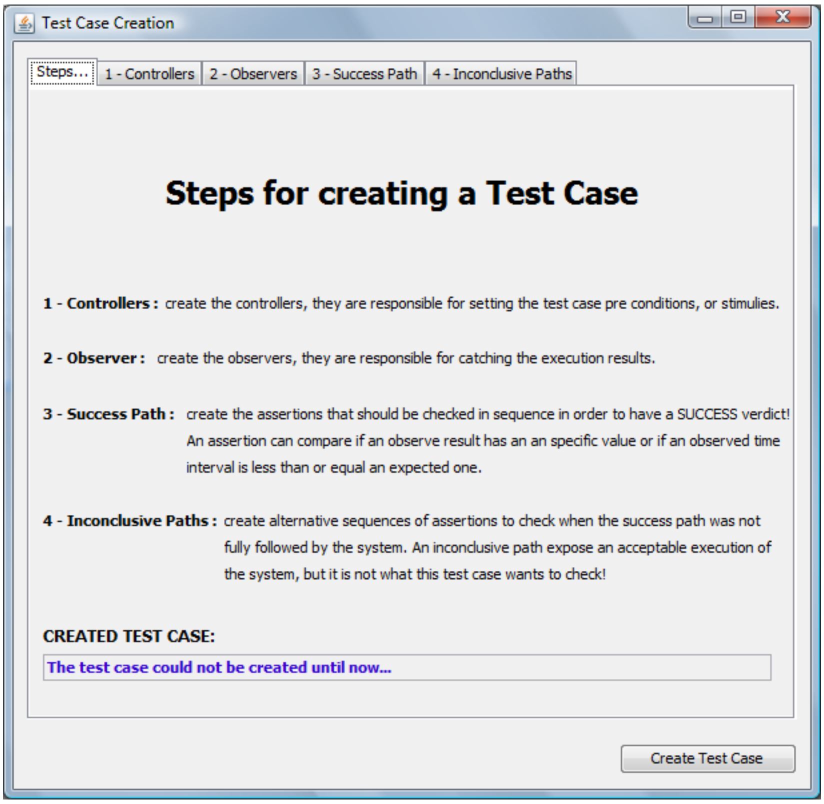
Figura 8.2: Testing Process

8.1.2 Case Study Definition

An initial infrastructure to support test execution in an actual real-time environment is investigated in [10]. However, the work presented in [90] extends the previous work by presenting an effective solution to support automation of test case execution for real-time systems.

The testing process considered in this case study is illustrated in Figure 8.2. It is divided into four well defined steps.

The first step is to define test cases according to the approach and theory presented in Chapters 5, 6, and 7. As there is a prototype tool implementing the proposed test case generation and selection strategy, the tester needs to manually instantiate a Java class to define the TIOSTS of the specification and the TIOSTS representing the test purpose. Once the specification and test purpose have been defined, the test case generation and selection is automatically performed.



Steps for creating a Test Case

1 - Controllers : create the controllers, they are responsible for setting the test case pre conditions, or stimulies.

2 - Observer : create the observers, they are responsible for catching the execution results.

3 - Success Path : create the assertions that should be checked in sequence in order to have a SUCCESS verdict! An assertion can compare if an observe result has an an specific value or if an observed time interval is less than or equal an expected one.

4 - Inconclusive Paths : create alternative sequences of assertions to check when the success path was not fully followed by the system. An inconclusive path expose an acceptable execution of the system, but it is not what this test case wants to check!

CREATED TEST CASE:

The test case could not be created until now...

Figura 8.3: Test Case Builder Application

At the second step, each test case is translated into C code. Another Java application has been implemented to support this activity as shown in Figure 8.3. Currently, the tester manually indicate all inputs and outputs of the generated test case in an interactive way and a C code implementing the test case is automatically generated. Furthermore, to make the system testable, a C API (see [90]) has been developed to instrument the code of the SUT. The SUT is instrumented for including Points of Control and Observation (PCOs). As the focus of this work is on functional testing, some examples of PCOs are the possibilities of observing values returned by functions, received messages, and timing associated with system responses. The instrumentation activity is manually performed by the tester using the implemented API.

The third step consists in the execution of the instrumented SUT guided by test cases. For this, a logging mechanism has been implemented in order to store all information needed to check the testing results. Considering that the SUT runs on a real-time environment such as a real-time operating system, it is important that the implementation logs its own information in order to reduce the number of processes and consequently avoid introduction of noise in the results. As we are dealing with RTES, each addition of code has a direct effect in the execution time of the application. Thus, test case execution can interfere with the flow of execution of the SUT, adding delays that can lead to false positives of failures. For minimizing this interference, all generated information is kept in the main memory. After the SUT execution, a simple text file is generated with the results. Logging frameworks such as log4c[1] are not suitable in this case since the quantity of dependencies forbid their execution within a dedicated hardware and the added code can cause a large delay at the actual execution time of the SUT.

The fourth step receives as input the text file with the execution results and provide verdicts for the tester. This step is not executed inside the execution platform level, but at the development platform level. An extension of the CUnit[2] framework has been developed for evaluating the text file with execution results and emit a verdict according to the test cases defined in the first step.

The case study was conducted by only one tester with large experience in TIOSTS models. Moreover, the case study aimed at testing only one scenario due to difficulties to completely execute the testing process. The scenario to be tested was based on a fault model profile based on common faults related to interruption testing [9] and potential faults in an implementation of a TIOSTS. The fault model given to the tester is specified in natural language and its description is defined as follows:

- After an interruption, the interrupted application does not maintain data previously received as input;
- After an interruption, the interrupted application does not continue its execution at the same point where it was interrupted;
- Unexpected outputs, when an implementation responds with an output not described in its specification;
- Clock guard restriction, when an implementation reduces an execution time range associated with an action;

[1] http://log4c.sourceforge.net

[2] http://cunit.sourceforge.net

- Clock guard widening, when an implementation increases an execution time range associated with an action.

It is important to remark that all information used as input for the execution of this case study was the high level description of the SUT presented in Section 4.3, the defined fault model, and the implementation of the SUT.

8.1.3 Case Study Results

This subsection presents and discusses the obtained results. The most critical scenario of the alarm system is the power failure exactly after some alarm has been triggered. In this case, the system must switch to backup power and continues its execution with the calling to the police and turning on the lights of the room where the sensor detected an intruder.

Considering the first activity of the testing process (see Figure 8.2), a TIOSTS specification was built along with a TIOSTS test purpose to check the defined scenario. Figures 8.4 and 8.5 present the defined specification and test purpose, respectively. However, all TIOSTS models automatically generated by the prototype tool are presented in Appendix B.

Table 8.1 summarizes the metrics collected during the execution of all activities of the test process. It is important to remark that the test case generation and implementation is performed in a different platform from the execution platform of the SUT. The development platform has the following characteristics: 2x3.00GHz CPU, 1536MB RAM, Ubuntu 10.10, CVC3 2.2, and UPPAAL DBM Library 2.0.7. The execution platform is based on FreeRTOS environment using the industrial PC (x86) port.

Considering the first activity of the testing process depicted in Figure 8.2, the tester spent 40 minutes to manually implement the TIOSTS representing the specification and 4 minutes to implement the TIOSTS test purpose (Table 8.1, lines 1 and 2, respectively). Once the specification and test purpose are built, the test case generation and selection is automatically performed in 3 seconds (Table 8.1, line 3).

The prototype tool generated 3 test cases (Table 8.1, line 4). Observing the specification presented in Figure 8.4 it is possible to realize that only considering the interruption after triggering an alarm, there are three possibilities. The first path is the scenario where a window breaking alarm is triggered (we denote TC1). At the second scenario, a door opening

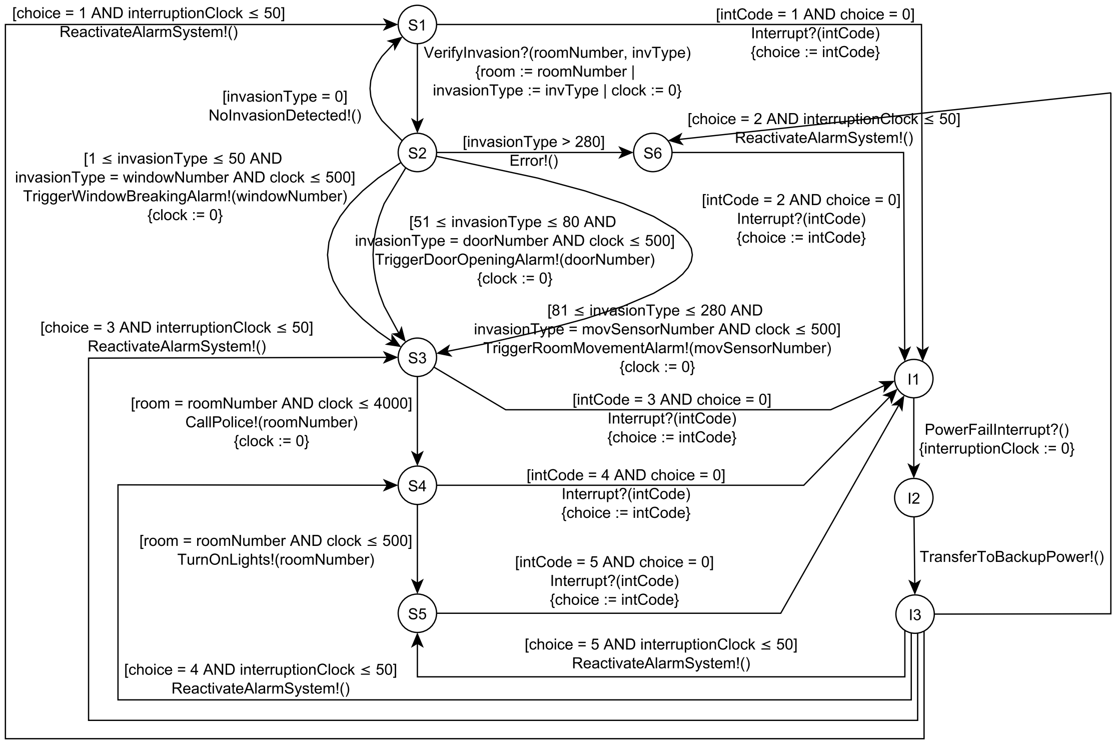

[choice = 1 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
S1
[intCode = 1 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
VerifyInvasion?(roomNumber, invType)
{room := roomNumber |
invasionType := invType | clock := 0}
[invasionType = 0]
NoInvasionDetected!()
[choice = 2 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
S2
[invasionType > 280]
Error!()
S6
[1 ≤ invasionType ≤ 50 AND
invasionType = windowNumber AND clock ≤ 500]
TriggerWindowBreakingAlarm!(windowNumber)
{clock := 0}
[intCode = 2 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
[51 ≤ invasionType ≤ 80 AND
invasionType = doorNumber AND clock ≤ 500]
TriggerDoorOpeningAlarm!(doorNumber)
{clock := 0}
[81 ≤ invasionType ≤ 280 AND
invasionType = movSensorNumber AND clock ≤ 500]
TriggerRoomMovementAlarm!(movSensorNumber)
{clock := 0}
[choice = 3 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
S3
I1
[room = roomNumber AND clock ≤ 4000]
CallPolice!(roomNumber)
{clock := 0}
[intCode = 3 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
PowerFailInterrupt?()
{interruptionClock := 0}
[intCode = 4 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
S4
I2
[room = roomNumber AND clock ≤ 500]
TurnOnLights!(roomNumber)
[intCode = 5 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
TransferToBackupPower!()
S5
I3
[choice = 5 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
[choice = 4 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()

Figura 8.4: TIOSTS Specification for the Burglar Alarm System Case Study

alarm is triggered (TC2). The last scenario is the case where a room movement alarm is triggered (TC3).

After the generation of test cases, the testing process is completely executed for each of them from the second step. Considering only the TC1, the next activity is to implement it and prepare the environment to the execution. For this, three activities were performed:

1. The test case builder application (Figure 8.3) was used to generate the C code implementing the test case in 25 minutes (Table 8.1, line 5). For this, the tester manually indicated all inputs, outputs, and time requirements of TC1. It is important to mention that all defined information (inputs, outputs, and so on) can be reused in the definition of other test cases with the same information and the CVC3 SMT Solver is used for automatically defining the inputs of the test case.

2. An interface was implemented to allow the communication between the SUT and test driver. This activity was performed in 18 minutes (Table 8.1, line 6) and it is used for all test cases.

3. Finally, the SUT was instrumented for the execution of TC1. 6 minutes were spent to

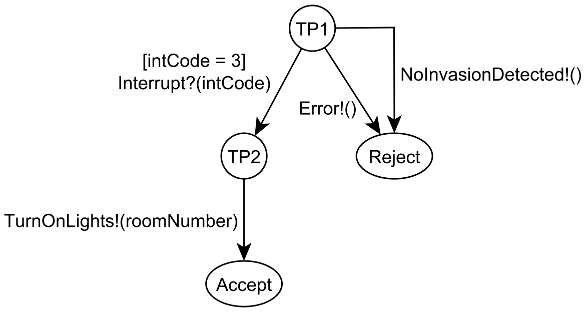
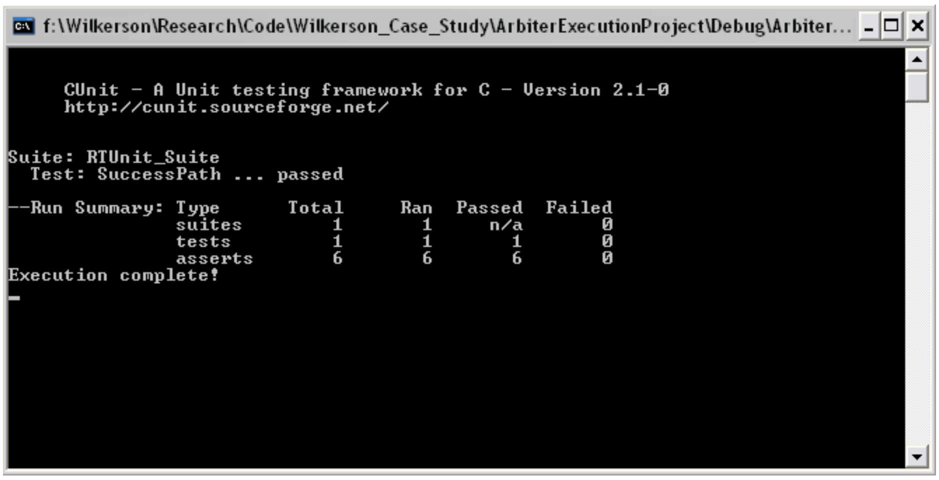


Figura 8.5: TIOSTS Test Purpose for the Burglar Alarm System Case Study


f:\Wilkerson\Research\Code\Wilkerson_Case_Study\ArbiterExecutionProject\Debug\Arbiter...

```
     CUnit - A Unit testing framework for C - Version 2.1-0
     http://cunit.sourceforge.net/


Suite: RTUnit_Suite
  Test: SuccessPath ... passed

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        1       1     n/a       0
               tests         1       1       1       0
               asserts       6       6       6       0
Execution complete!
```

Figura 8.6: Results of the TC1 Execution

perform this activity (Table 8.1, line 7).

The next step is to deploy the instrumented SUT, execute it to extract the generated log file, and evaluate the results. As this evaluation is automatically performed (see Figure 8.6) it takes little time (Table 8.1, line 8).

Considering the execution of TC2 (Table 8.1, lines from 9 to 11) and TC3 (Table 8.1, lines from 12 to 14), the testing process activities take less time than the activities related to the execution of TC1 because many information can be reused.

Since all needed information was collected, the questions of the defined measurement model can be answered. For the first question, "what is the effort required to use this approach?", we have the following effort required to test the defined scenario considering all

Tabela 8.1: Metrics of the Burglar Alarm System Case Study

	Metrics	**Time**
1	Definition and implementation of the TIOSTS specification	40 *min*
2	Definition and implementation of the TIOSTS test purpose	4 *min*
3	Test case generation time	3 *sec*
4	Number of Test Cases	3
5	Implementation of TC1	25 *min*
6	Implementation of an interface to allow the communication between SUT and test driver	18 *min*
7	Instrumentation of SUT for executing TC1	6 *min*
8	Evaluation of results for TC1	1 *sec*
9	Implementation of TC2	5 *min*
10	Instrumentation of SUT for executing TC2	5 *min*
11	Evaluation of results for TC2	1 *sec*
12	Implementation of TC3	5 *min*
13	Instrumentation of SUT for executing TC3	5 *min*
14	Evaluation of results for TC3	1 *sec*
15	Execution time of all test process for all generated test cases	113.1 *min*
16	Fault model coverage	100%
17	Number of invalid TCs	0 (0%)

generated test cases:

$$\begin{aligned} E &= E_1 + E_2 + E_3 + E_4 + E_5 \\ &= 40\ \mathit{min} + 4\ \mathit{min} + 3\ \mathit{sec} + 69\ \mathit{min} + 3\ \mathit{sec} \\ &= 113.1\ \mathit{min} \end{aligned}$$

The fault model profile defined in Subsection 8.1.2 was instantiated in order to evaluate the effectiveness of the approach proposed in this thesis. Thus, the following real defects were inserted in the SUT:

- After the power failure interruption, the calling to the police informs a wrong room number;

- After the power failure interruption, the SUT turns on the lights of the room where the sensor detected an intruder instead of calling to the police;
- After calling to the police, the SUT performs an unspecified output;
- The SUT performs the action of turning on the lights in more than 50 ms.

As the specification does not specify lower bounds as time requirements, it not possible to insert faults related to clock guard restrictions. As shown in Table 8.1, line 16, 100% of the defined fault model instance is covered by the defined test cases.

Finally, as all generated test cases were executed, no invalid test cases could be detected (Table 8.1, line 17).

8.2 The Automatic Guided Vehicle System

The automatic guided vehicle (AGV) system consists of a robot for autonomous navigation that is able to plan and execute predefined tasks. Automatic guided vehicles are used, for example, to transport materials in industries, for inspection in risk areas or deposits of toxic materials, etc.

Basically, the first step of an AGV system is to define the plan to be followed. After that, the execution is started. Sensors are used to drive the vehicle and allow it to overcome obstacles of the path. Deviations from the original path may be required because of obstacles and avoid collisions.

This case study is aimed at generating test cases for the AGV system. Test execution is not considered because we do not have an implementation of this system. Even thus, the AGV case study allows us to evaluate the test case generation activity including the generation of test cases for checking interruptions.

8.2.1 The GQM Measurement Model

Since test execution is not considered in this case study, another measurement model was defined (Figure 8.7). The goal is the same as in Section 8.1, but only two questions were defined to characterize the measurement objects:

What is the effort required to use this approach? This question is intended to evaluate the effort required to apply all the test process from the building of the model to test case generation. For answering this question the following metric was defined: $E = E_1 + E_2 + E_3$, where:

- E_1 is the time spent to build the model using the TIOSTS formalism;
- E_2 is the time spent to define test purposes in order to test specific scenarios.
- E_3 is the time spent to generate test cases using the developed prototype tool.

How effective is this approach? The objective of this question is to evaluate the effectiveness of the proposed approach w.r.t. fault coverage. Thus, the C metric indicates the ability of generated test cases for uncovering faults described by a previously defined fault model.

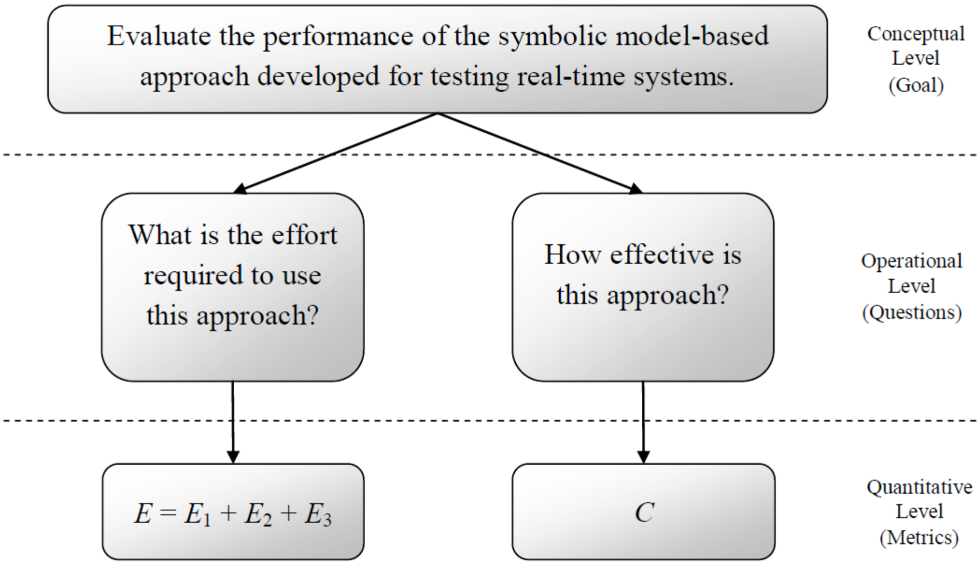

Evaluate the performance of the symbolic model-based approach developed for testing real-time systems.
Conceptual Level (Goal)
What is the effort required to use this approach?
How effective is this approach?
Operational Level (Questions)
$E = E_1 + E_2 + E_3$
C
Quantitative Level (Metrics)

Figura 8.7: Measurement Model for the AGV Case Study

8.2.2 Case Study Definition

The testing process considered in this case study is composed of three steps: build the model, define test purposes, and generate test cases. The first step is related to the definition of a TIOSTS representing the specification. Next, at the second step, test purposes are defined also using the TIOSTS formalism. Once the specification and test purposes have been defined, the test case generation and selection are automatically performed by the developed prototype tool.

The case study was conducted by only one tester with large experience in TIOSTS models. The case study aimed at testing two scenarios: an execution with no interruptions and another with one interruption. The first scenario was chosen for demonstrating that the strategy can be used for generating test cases with no interruptions. The latter is related to the most important scenario: an interruption occurs because an obstacle has been identified.

All information used as input for the execution of this case study was a high level description of the SUT described in two pages and the fault model defined in Subsection 8.1.2.

8.2.3 Case Study Results

This subsection presents and discusses the obtained results. Considering the first activity of the process described in Subsection 8.2.2, a TIOSTS representing the specification was defined (see Figure 8.8). At the beginning, the AGV system expects as input (represented by the action between locations $S1$ and $S2$ in Figure 8.8) the initial reference and path of the plan to be followed. After the input action, the AGV system emits several output actions: a message indicating that the file was successfully read (action between locations $S2$ and $S3$), another indicating that the references were successfully decoded (action between locations $S3$ and $S4$), and another message indicating that the path was successfully decoded (action between locations $S4$ and $S5$). After reading all needed information, the AGV system starts moving (action between locations $S5$ and $S6$). A periodic task is executed every 2000 milliseconds when the AGV system is moving. For controlling this task the *periodicClock* clock is used. Moreover, two interruptions can occur when the AGV system is moving: one related to the self diagnosis (Locations $I2.1$ and $I2.2$) and another related to the detection of an obstacle (Locations from $I1.1$ to $I1.4$). The *interruptionClock* clock is used to indicate that the latter interruption must be treated within at most 500 milliseconds. Finally, the AGV system emits an output message to indicate that the plan was successfully executed.

The test purpose depicted in Figure 8.9, named TP1, was defined for testing the scenario where no interruptions occur and the other test purpose of Figure 8.10, named TP2, was defined for testing the scenario where an obstacle is identified and an interruption occurs. All

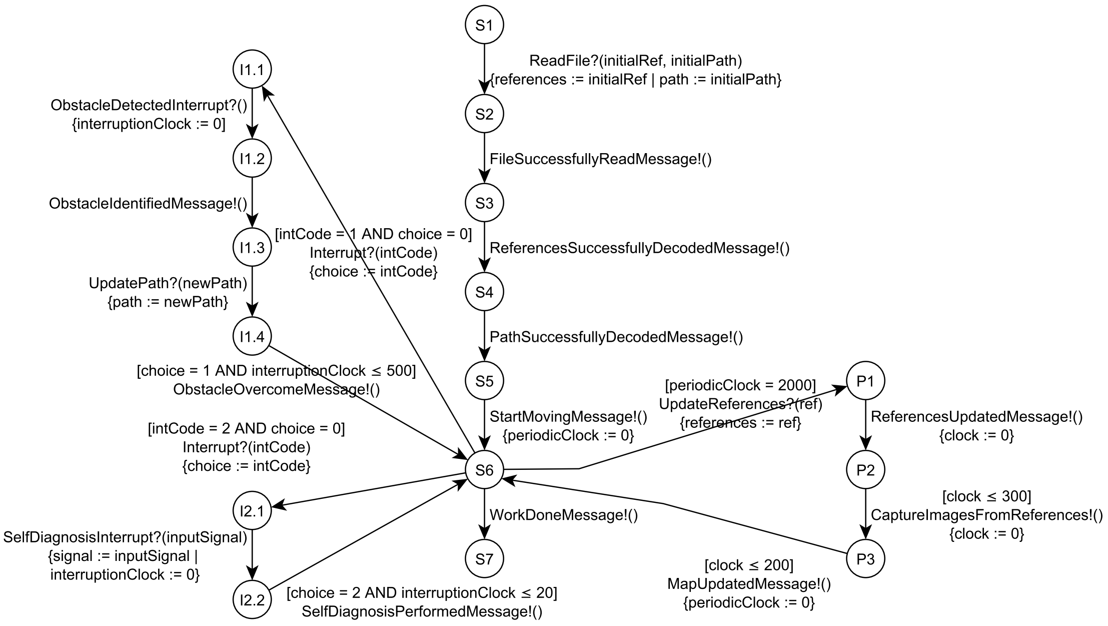

S1
ReadFile?(initialRef, initialPath)
{references := initialRef | path := initialPath}
S2
FileSuccessfullyReadMessage!()
S3
ReferencesSuccessfullyDecodedMessage!()
S4
PathSuccessfullyDecodedMessage!()
S5
StartMovingMessage!()
{periodicClock := 0}
S6
WorkDoneMessage!()
S7
I1.1
ObstacleDetectedInterrupt?()
{interruptionClock := 0]
I1.2
ObstacleIdentifiedMessage!()
I1.3
UpdatePath?(newPath)
{path := newPath}
I1.4
[intCode = 1 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
[choice = 1 AND interruptionClock ≤ 500]
ObstacleOvercomeMessage!()
[intCode = 2 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
I2.1
SelfDiagnosisInterrupt?(inputSignal)
{signal := inputSignal |
interruptionClock := 0}
I2.2
[choice = 2 AND interruptionClock ≤ 20]
SelfDiagnosisPerformedMessage!()
[periodicClock = 2000]
UpdateReferences?(ref)
{references := ref}
P1
ReferencesUpdatedMessage!()
{clock := 0}
P2
[clock ≤ 300]
CaptureImagesFromReferences!()
{clock := 0}
P3
[clock ≤ 200]
MapUpdatedMessage!()
{periodicClock := 0}

Figura 8.8: TIOSTS Specification for the AGV Case Study

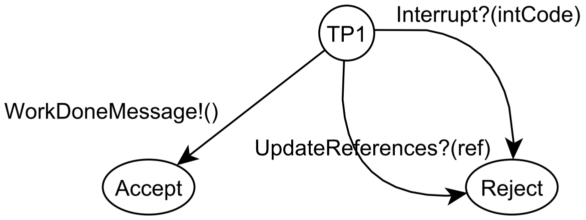


Figura 8.9: TIOSTS Test Purpose for the Scenario with no Interruptions (TP1)

TIOSTS models automatically generated by the prototype tool are presented in Appendix B.

Table 8.2 summarizes the metrics collected during the execution of all activities defined for this case study. Considering the first activity of the process, the tester spent 50 minutes to manually implement the TIOSTS representing the specification (Table 8.2, line 1).

Considering the scenario with no interruptions, the test purpose was defined in 4 minutes (Table 8.2, line 2). Once the specification and test purpose is implemented, the test case generation and selection was automatically performed in 3 seconds (Table 8.2, line 3) and only one test case was generated (Table 8.2, line 4).

The test purpose for the scenario with one interruption was defined in 3 minutes (Table 8.2, line 5). For this case, the test case generation and selection was performed in 3 seconds (Table 8.2, line 6) and only one test case was generated (Table 8.2, line 7).

Since all needed information was collected, the questions of the defined measurement

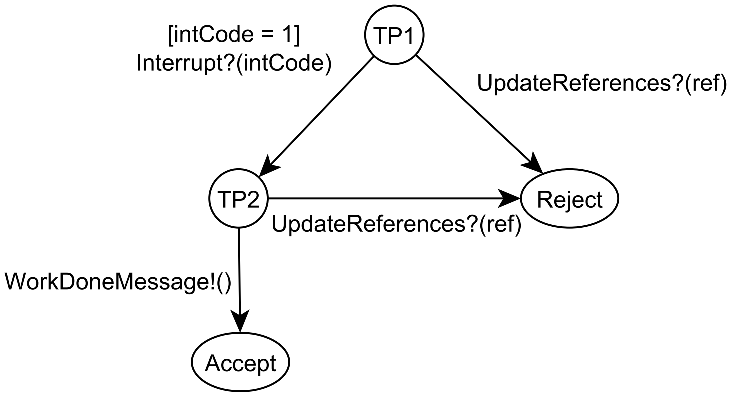


Figura 8.10: TIOSTS Test Purpose for the Scenario with One Interruption (TP2)

Tabela 8.2: Metrics of the AGV Case Study

	Metrics	**Time**
1	Definition and implementation of the TIOSTS specification	50 *min*
2	Definition and implementation of the TIOSTS test purpose TP1	4 *min*
3	Test case generation time considering TP1	3 *sec*
4	Number of Test Cases considering TP1	1
5	Definition and implementation of the TIOSTS test purpose TP2	3 *min*
6	Test case generation time considering TP2	3 *sec*
7	Number of Test Cases considering TP2	1
8	Execution time of all test process for all generated test cases	57.1 *min*
9	Fault model coverage	100%

model can be answered. For the first question, “what is the effort required to use this approach?”, we have the following effort required to generate test cases for the two defined scenarios:

$$\begin{aligned} E &= E_1 + E_2 + E_3 \\ &= 50\,\mathit{min} + 4\,\mathit{min} + 3\,\mathit{min} + 3\,\mathit{sec} + 3\,\mathit{sec} \\ &= 57.1\,\mathit{min} \end{aligned}$$

The fault model profile defined in Subsection 8.1.2 was instantiated, based on the scenarios described in Subsection 8.2.2, in order to evaluate the effectiveness of the proposed approach.

- After the obstacle detection interruption, the AGV system does not maintain the data of the path to be followed;
- After the obstacle detection interruption, the AGV system does not finish its mission;
- After receiving initial references and the initial path, the SUT performs an unspecified output;
- An obstacle is overcome in more than 500 ms.

As the specification does not specify lower bounds as time requirements, it not possible to instantiate faults related to clock guard restrictions. As shown in Table 8.2, line 9, 100% of the defined fault model instance is covered by the defined test cases.

8.3 Concluding Remarks

This chapter presented two case studies performed to evaluate the applicability of the proposed approach. The first case study allowed to evaluate the test case generation and execution once an implementation is available. It was not possible to execute test cases for the second case study because there is no implementation available. Even thus, these case studies allowed to realize that the effort spent to generate test cases for checking specific scenarios is minimal when compared to the time spent to perform the entire process, even when interruptions are taken into account. Another strength of the work is the automation of some parts of the test process such as test case generation and evaluation of the results given that the logs have been generated. However, some points of improvements were identified such as the development of algorithms to translate the TIOSTS test case into C code in order to reduce the time spent to execute test cases.

Capítulo 9

Considerações Finais

Este capítulo resume os principais resultados deste trabalho e apresenta as sugestões de trabalhos futuros.

9.1 Conclusions

The main objective of this thesis was to provide an approach to conformance testing of real-time systems based on the use of a symbolic model that abstracts both time and data in order to broadening the application of conformance testing in this field. This thesis also presented a conformance testing theory to deal with the model proposed and described how test cases can be generated. Moreover, interruption testing of real-time systems was taken into account. For this, as a result of an initial investigation, an approach to conformance testing of non-real-time reactive systems with interruptions was proposed.

Considering the research questions defined in Chapter 1, the following results were achieved:

Research Question 1 *In which ways can we extend the symbolic model-based testing theory to be able to test real-time systems in an accurate manner?*

In order to answer this first research question a new symbolic model-based testing approach was proposed by combining symbolic transition systems [108] with timed automata [5]. Thus, the proposed model can handle both data and time requirements symbolically (see

Chapter 5). Furthermore, a conformance testing framework is proposed in Chapter 6 along with algorithms for test case generation.

Research Question 2 *In a real-time symbolic model-based testing context, how can we provide models to be able to specify and test asynchronous events such as interruptions in an accurate manner?*

In order to answer this second research question an initial investigation was performed in the context of non-real-time reactive systems. As a result, Chapter 3 presented a complete conformance testing approach for reactive systems. This work intended to investigate interruptions in a simple context. Considering the interruption testing of real-time systems, Chapter 7 presented a strategy based on the proposed symbolic model-based conformance testing approach to specify and test interruptions.

Research Question 3 *In a real-time symbolic model-based testing context, is it possible to provide an automated oracle?*

As discussed in Chapter 2, an oracle is a mechanism composed of a result generator and a comparator. In an automated oracle these two activities are fully automated. As the test cases generated by our approach describes all outputs allowed by specifications, the first activity of an oracle is automatically performed. According to what was discussed in Section 6.3, each execution of an implementation produces a log that describes the executed scenario. Next, this log is compared to the test case in order to emit a verdict. As this comparison is automatically performed (see Chapter 8), we can state that an automated oracle was provided by our approach. But it is important to remark that the test process is not completely automated, since TIOSTS test cases are manually translated into C code.

In summary, this thesis provides the following contributions:

1. An approach to conformance testing of reactive systems with interruptions and a tool to facilitate the practical application of the proposal;

2. A complete review of relevant work on conformance testing of real-time systems, described in Chapter 4, that resulted in the identification of some open problems.

3. A new conformance testing approach to real-time systems, where the SUT is modelled using a symbolic model that abstract both time and data;
4. A test case generation process based on symbolic execution and constraint solving for the data aspects combined with symbolic analysis of timed aspects.
5. A prototype tool implementing all algorithms of the test case generation process, which is essential in the generation of test cases from symbolic models.
6. A strategy to interruption testing of real-time systems along with a way of defining test purposes in order to check specific interruptions;
7. An initial test architecture including automatic ways of test execution and reliable verdicts achievements.
8. Results of case studies involving the use of the proposed work that show the feasibility of the practical application of the proposal.

Considering the related work presented in Chapter 4, all tables are presented here again in order to compare the related work with our approach. Table 9.1 shows that our approach generates test cases in an offline way using test purposes as test case selection strategy, there is tool support based on TIOSTS specification language, and quiescence is not taken into account. Table 9.2 shows that the tioco conformance relation is adopted and only deterministic specifications are taken into account along with input-complete implementations. Table 9.3 presents that our approach deals with analogue-time models and it is able to generate both instantiated and abstract test cases, considering that the CVC3 SMT Solver can be used to instantiate abstract test cases. Furthermore, our approach allows the specification of synchronous and asynchronous events and provides an automated oracle.

9.2 Future Work

With the completion of this work, there are several opportunities for future work. Next, some ideas are described:

Work	**Test Case Generation**	**TP**	**Tool**	**Spec. Language**	**Quiesc.**
Cardell-Oliver	offline	yes*	Essex*	TIOLTS	no
En-Nouaary et al.	offline	yes	no	deterministic and output urgent TAIO	no
Li et al.	offline	yes	no	RT Statecharts	no
Khoumsi	offline	yes	no	non-deterministic TIOSA	no
Briones and Brinksma	offline	no	no	TIOLTS	yes
Bohnenkamp and Belinfante	online	yes	yes*	non-deterministic safety TAIO	yes
Bodeveix et al.	offline	yes	no	a kind of TAIO	no
Larsen et al.	online	yes	TRON	TAIO (with guards on locations and transitions)	no
Hessel et al.	offline	yes	CoVer	deterministic and output urgent TAIO	no
Merayo et al.	offline	no	no	non-deterministic TEFSM	no
Krichen and Tripakis	offline and online	yes	TTG*	partially-observable and non-deterministic TAIO	no
Zheng et al.	offline	yes*	TROMLAB*	TEFSM	no
David et al.	offline	yes	TIGA	TIOGA	no
Adjir et al.	offline	yes	TINA	Prioritized Time Petri Nets	no
Styp et al.	no	no	no	STA	no
Timo et al.	offline	yes	no	VDTA	no
Andrade	offline	yes	yes	TIOSTS	no

Tabela 9.1: Comparison with Related Work

Take quiescence into account: in practice, tests observe the behaviour of the system and the absence of outputs. Then, an important future work is to extend the proposed conformance testing approach to deal with quiescence;

Take internal actions into account: although the TIOSTS definition allows the specification of internal actions, the developed algorithms do not treat internal actions;

Deal with non-input-complete implementations: practically all reviewed approaches assume the input-completeness of the implementation. the intention of this future work is to investigate how this assumption could be discarded;

Deal with non-deterministic models: the intention here is to extend the approach proposed in this thesis to deal with non-deterministic models;

Propose generic fault models: fault models based on approaches that use region clocks such as the work presented in [48; 1] cannot be completely reused in a context where zones are used because of the higher abstraction level. Then, this issue must be better investigated;

Generate test cases based on coverage criteria: this thesis proposes a test selection approach based on test purposes, but there are other approaches to test case selection such as selection based on coverage criteria;

Deal with more data types: investigate techniques such as abstract interpretation in order to deal with more data types;

Improve the test execution activity: the test case execution can be improved by automating the translation of TIOSTS test cases into C code. In this case, the time spent in the test case execution activity can be decreased;

Execution of test cases in other platforms: the intention is to extend the work proposed in this thesis to allow the execution of test cases in other environments besides FreeRTOS;

Develop a fully integrated environment for testing real-time systems: it is important to integrate the test case generation tool with the test execution tools in order to provide a complete environment to generate and execute test cases;

Perform new case studies: since a complete environment for test case generation and execution is proposed as a future work, it is essential to perform new case studies in order to evaluate the applicability of the work.

Work	Conf. Relation	Specification	Implementation
Cardell-Oliver	trace equivalence	input-complete and must have more states than the implementation.	input-complete
En-Nouaary et al.	trace equivalence	input-complete and must have the same number of locations as the implementation.	input-complete
Li et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Khoumsi	timed trace inclusion	input-complete	input-complete
Briones and Brinksma	ioco with quiescence	input-complete	input-complete*
Bohnenkamp and Belinfante	ioco with quiescence	input-complete	input-complete
Bodeveix et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Larsen et al.	timed trace inclusion	deterministic and input-complete	input-complete
Hessel et al.	timed trace inclusion	deterministic, input-complete, and output urgent	input-complete
Merayo et al.	there are several conformance relations	input-complete	input-complete
Krichen and Tripakis	tioco	no restriction on input-completeness	input-complete
Zheng et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
David et al.	tioco	input-complete	input-complete
Adjir et al.	timed trace inclusion	deterministic, input-complete, and output urgent	deterministic, input-complete, and output urgent
Styp et al.	stioco	non-deterministic	input-complete
Timo et al.	tvco	assumptions are not discussed	assumptions are not discussed
Andrade	tioco	deterministic and no restriction on input-completeness	input-complete

Tabela 9.2: Comparison with Related Work

Work	Time	Test Cases	Communication	Oracle
Cardell-Oliver	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
En-Nouaary et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Li et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Khoumsi	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Briones and Brinksma	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Bohnenkamp and Belinfante	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Bodeveix et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Larsen et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Hessel et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Merayo et al.	digital-time model	instantiated	synchronous	partial
Krichen and Tripakis	digital and analogue-time models*	instantiated	synchronous	automated
Zheng et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
David et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Adjir et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Styp et al.	analogue-time model	undefined	synchronous	undefined
Timo et al.	analogue-time model	undefined	synchronous	undefined
Andrade	analogue-time model	instantiated and abstract	synchronous and asynchronous	automated

Tabela 9.3: Comparison with Related Work

Bibliografia

[1] M. S. AbouTrab and S. Counsell. Fault coverage measurement of a timed test case generation approach. In *ECBS '10: Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 141–149, Washington, DC, USA, 2010. IEEE Computer Society.

[2] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[3] Noureddine Adjir, Pierre Saqui-Sannes, and Kamel Mustapha Rahmouni. Testing real-time systems using tina. In *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, TESTCOM '09/FATES '09, pages 1–15, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Noureddine Adjir, Pierre Saqui-Sannes, and Kamel Mustapha Rahmouni. Time-optimal real-time test case generation using prioritized time petri nets. In *VALID '09: Proceedings of the First International Conference on Advances in System Testing and Validation Lifecycle*, pages 110–116, 2009.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[6] Wilkerson L. Andrade. Interaction test case generation for mobile phone applications. Master's thesis, Federal University of Campina Grande, Mar 2007.

[7] Wilkerson L. Andrade and Patrícia D. L. Machado. Modeling and testing interruptions in reactive systems using symbolic models. In *SAST'08: Proc. of the 2nd Brazilian*

Work. on Systematic and Automated Software Testing, pages 34–43, Porto Alegre, 2008. SBC.

[8] Wilkerson L. Andrade and Patrícia D. L. Machado. Interruption testing of reactive systems. In *Formal Methods: Foundations and Applications*, volume 5902 of *LNCS*, pages 37–53. Springer, 2009.

[9] Wilkerson L. Andrade and Patrícia D. L. Machado. Interruption testing of reactive systems. *Formal Aspects of Computing*, pages 1–23, 2011. To appear.

[10] Wilkerson L. Andrade, Patrícia D. L. Machado, Everton L. G. Alves, and Diego R. Almeida. Test case generation of embedded real-time systems with interruptions for FreeRTOS. In *Formal Methods: Foundations and Applications*, volume 5902 of *LNCS*, pages 54–69. Springer, 2009.

[11] Wilkerson L. Andrade, Patrícia D. L. Machado, Thierry Jéron, and Hervé Marchand. Abstracting time and data for conformance testing of real-time systems. In *A-MOST '11: Proceedings of the 7th Workshop on Advances in Model Based Testing*, March 2011. To appear.

[12] Wilkerson L. Andrade, Francisco G. O. Neto, and Patrícia D. L. Machado. Geração de casos de teste de interrupção para aplicações de celulares. In *WTF '07: Proc. of the VIII Test and Fault Tolerance Workshop*, pages 129–142, Porto Alegre, RS, Brazil, 2007. Brazilian Computer Society.

[13] George S. Avrunin, James C. Corbett, and Laura K. Dillon. Analyzing partially-implemented real-time systems. *IEEE Trans. Softw. Eng.*, 24(8):602–614, 1998.

[14] Victor R. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, University of Maryland at College Park, College Park, MD, USA, 1992.

[15] Gilles Bernot. Testing against formal specifications: a theoretical view. In *TAPSOFT '91: Vol. 2*, pages 99–119, New York, NY, USA, 1991. Springer-Verlag.

[16] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, Mar 1991.

[17] N. Bertrand, T. Jéron, A. Stainer, and M. Krichen. Off-line test selection with test purposes for non-deterministic timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, march 2011. To appear.

[18] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, Boston, MA, USA, 1999.

[19] Jean-Paul Bodeveix, Rachid Bouaziz, and Ousmane Koné. Test method for embedded real-time systems. In *ERCIM European Workshop on Dependable Software Intensive Embedded Systems*, pages 1–10, Porto, Portugal, 2005. ERCIM.

[20] Henrik Bohnenkamp and Axel Belinfante. Timed testing with TorX. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188, Newcastle, UK, 2005. Springer-Verlag.

[21] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 264–279. Springer, 1998.

[22] Ahmed Bouajjani, Yassine Lakhnech, and Sergio Yovine. Model-checking for extended timed temporal logics. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 306–326, London, UK, 1996. Springer-Verlag.

[23] Laura Brandán Briones and Ed Brinksma. A test generation framework for *quiescent* real-time systems. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 64–78. Springer, 2005.

[24] Laura Brandán Briones and Ed Brinksma. Testing real-time multi input-output systems. In *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 264–279. Springer-Verlag, 2005.

[25] Renée C. Bryce and Charles J. Colbourn. Test prioritization for pairwise interaction coverage. In *A-MOST '05: Proceedings of the first international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[26] Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, 2008.

[27] Jens R. Calamé, Natalia Ioustinova, and Jaco van de Pol. Automatic model-based generation of parameterized test cases using data abstraction. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proc. of the Doctoral Symposium affiliated with the Fifth Integrated Formal Methods Conference (IFM 2005)*, volume 191 of *Electronic Notes in Computer Science*, pages 25–48. Elsevier, October 2007.

[28] Rachel Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, Dec. 2000.

[29] Emanuela G. Cartaxo, Wilkerson L. Andrade, Francisco G. O. Neto, and Patrícia D. L. Machado. LTSBT: A tool to generate and select functional test cases for embedded systems. In *SAC'08: Proc. of the 2008 ACM symposium on Applied computing*, volume 2, pages 1540–1544, New York, NY, USA, 2008. ACM Press.

[30] Emanuela G. Cartaxo, Patrícia D. L. Machado, and Francisco G. Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 2009. Early Online View: http://dx.doi.org/10.1002/stvr.413.

[31] Albert M. K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[32] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In *TACAS'02: Proc. of the Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 151–173. Springer, 2002.

[33] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[34] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Camille Constant, Thierry Jéron, Hervé Marchand, and Vlad Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Trans. Software Eng.*, 33(8):558–574, 2007.

[36] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[37] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. A game-theoretic approach to real-time system testing. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 486–491, New York, NY, USA, 2008. ACM.

[38] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Timed testing under partial observability. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 61–70, Washington, DC, USA, 2009. IEEE Computer Society.

[39] André L. L. de Figueiredo, Wilkerson L. Andrade, and Patrícia D. L. Machado. Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006. Proceedings of the AMOST'2006.

[40] R. G. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.

[41] R. G. de Vries and J. Tretmans. Towards formal test purposes. In *Proceedings of 1st International Workshop on Formal Approaches to Testing of Software 2001 (FATES'01)*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denmark, August 2001.

[42] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1990.

[43] I. K. El-Far and J. A. Whittaker. Model-based software testing. *Encyclopedia on Software Engineering*, 2001.

[44] Abdeslam En-Nouaary. A scalable method for testing real-time systems. *Software Quality Control*, 16:3–22, March 2008.

[45] Abdeslam En-Nouaary and Rachida Dssouli. A guided method for testing timed input output automata. In Dieter Hogrefe and Anthony Wiles, editors, *Testing of Communicating Systems*, volume 2644 of *Lecture Notes in Computer Science*, pages 211–225. Springer Berlin / Heidelberg, 2003. Proceedings of the 15th IFIP international conference on Testing of communicating systems (TestCom'03).

[46] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 28(11):1023–1038, 2002.

[47] Abdeslam En-Nouaary, Rachida Dssouli, Ferhat Khendek, and Abdelkader Elqortobi. Timed test cases generation based on state characterization technique. In *RTSS '98: Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 220–230, Washington, DC, USA, 1998. IEEE Computer Society.

[48] Abdeslam En-Nouaary, Ferhat Khendek, and Rachida Dssouli. Fault coverage in testing real-time systems. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 150, Washington, DC, USA, 1999. IEEE Computer Society.

[49] ETSI. *European Standard (ES) 201 873 - The Testing and Test Control Notation Version 3 (TTCN-3), Part 1: TTCN-3 Core Language, Part 2: Tabular Presentation Format for TTCN-3 (TFT), Part 3: Graphical Presentation Format for TTCN-3 (GFT), Part 4: Operational Semantics, Part 5: The TTCN-3 Runtime Interface (TRI), Part*

6: The TTCN-3 Control Interfaces (TCI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 2005.

[50] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property oriented test case generation. In *Formal Approaches to Software Testing, Proceedings of FATES 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 147–163, Montreal, Canada, 2004. Springer.

[51] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, 1997.

[52] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer, 2006.

[53] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *Testing of Communicating Systems*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

[54] L.D. Gowen. Specifying and verifying safety-critical software systems. *Proceedings of the IEEE Seventh Symposium on Computer-Based Medical Systems*, pages 235–240, Jun 1994.

[55] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer, 1993.

[56] Dick Hamlet. Software quality, software process, and software testing. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 41, pages 191–229. Academic Press, 1995.

[57] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, 2004.

[58] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111:193–244, June 1994.

[59] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M.

Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 77–117. Springer, 2008.

[60] Anders Hessel, Kim Guldstrand Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using UPPAAL. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *LNCS*, pages 114–130. Springer, 2004.

[61] Anders Hessel and Paul Pettersson. A test case generation algorithm for real-time systems. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 268–273, Washington, DC, USA, 2004. IEEE Computer Society.

[62] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[63] Bertrand Jeannet, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Symbolic test selection based on approximate analysis. In *TACAS'05: Proc. of Int. Conf. on Tools and Alg. for Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 349–364, 2005.

[64] Thierry Jéron. Symbolic model-based test selection. *Electron. Notes Theor. Comput. Sci.*, 240:167–184, July 2009. Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008).

[65] Thierry Jéron, Hervé Marchand, and Vlad Rusu. Symbolic determinisation of extended automata. In *Proceedings of the 4th IFIP International Conference on Theoretical Computer Science*, volume 209 of *IFIP book series*, pages 197–212. Springer-Verlag, 2006.

[66] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving. In *ICST '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, pages 479–488, Washington, DC, USA, 2010. IEEE Computer Society.

[67] Paul Jorgensen. *Software Testing: A Craftman's Approach*. CRC Press, Inc., 3rd edition, 2008.

[68] D. S. Jovanovic, B. Orlic, and J. F. Broenink. On issues of constructing an exception handling mechanism for CSP-based process-oriented concurrent software. In *Proceedings of Communicating Process Architectures CPA 2005*, pages 18–21, Eindhoven, NL, 2005. IOS Press.

[69] Ahmed Khoumsi. Complete test graph synthesis for symbolic real-time systems. *ENTCS*, 130:79–100, 2005.

[70] Ahmed Khoumsi. On synthesizing test cases in symbolic real-time testing. *Journal of the Brazilian Computer Society*, 12:31–48, 2007.

[71] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[72] Pieter Koopman and Rinus Plasmeijer. Testing reactive systems with GAST. In Stephen Gilmore, editor, *Trends in Functional Programming*, volume 4 of *Trends in Functional Programming*, pages 111–129. Intellect, 2003.

[73] Moez Krichen. *Model-Based Testing for Real-Time Systems*. PhD thesis, Université Joseph Fourier, Dec 2007.

[74] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *SPIN'04: Proc. of the 11th Int. SPIN Workshop on Model Checking of Software*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004.

[75] Moez Krichen and Stavros Tripakis. Real-time testing with timed automata testers and coverage criteria. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 134–151. Springer-Verlag, 2004.

[76] Moez Krichen and Stavros Tripakis. An expressive and implementable formal framework for testing real-time systems. In *TestCom'05: Proc. of the 17th IFIP Int. Conf. on Testing of Communicating Systems*, volume 3502 of *LNCS*, pages 209–225. Springer, 2005.

[77] Moez Krichen and Stavros Tripakis. Interesting properties of the real-time conformance relation tioco. In *ICTAC'06: Proc. of the 3rd Int. Colloquium on Theoretical Aspects of Computing*, volume 4281 of *LNCS*, pages 317–331. Springer, 2006.

[78] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[79] Phillip A. Laplante. *Real-Time System Design and Analysis*. John Wiley & Sons, 2004.

[80] Kim Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 79–94. Springer, 2005.

[81] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM.

[82] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[83] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes: Adapting the notion of specification to testing. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, pages 127–134, Washington, DC, USA, 2001. IEEE Computer Society.

[84] Daniel Leitao, Dante Torres, and Flávia Barros. NLForSpec: Translating natural language descriptions into formal test case specifications. In *Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2007)*, pages 129–134, Boston, Massachusetts, USA, 2007. Knowledge Systems Institute Graduate School.

[85] Grégory Lestiennes and Marie-Claude Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *ISSRE'02: Proc. of the 13th Int. Symp. on Software Reliability Engineering*, page 3. IEEE Computer Society, 2002.

[86] Qing Li and Carolyn Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

[87] Shuhao Li, Ji Wang, Wei Dong, and Zhi-Chang Qi. Property-oriented testing of real-time systems. In *APSEC'04: Proc. of the 11th Asia-Pacific Software Engineering Conference*, pages 358–365. IEEE Computer Society, 2004.

[88] L. Lorentsen, A.-P. Tuovinen, and J. Xu. Modelling feature interactions in mobile phones. In *Feature Interaction in Composed Systems (ECOOP 2001)*, pages 7–13, Budapest, Hungary, 2001.

[89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, sep 1989.

[90] Augusto Q. Macedo, Wilkerson L. Andrade, Diego R. Almeida, and Patrícia D. L. Machado. Automating test case execution for real-time embedded systems. In *ICTSS'10: Proceedings of the 22nd IFIP International Conference on Testing Software and Systems*, pages 37–42, 2010. Short Paper.

[91] Patrícia D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, LFCS, University of Edinburgh, UK, 2000.

[92] Patrícia D. L. Machado and Wilkerson L. Andrade. The oracle problem for testing against quantified properties. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pages 415–418, Washington, DC, USA, 2007. IEEE Computer Society.

[93] Patrícia D. L. Machado and Augusto C. A. Sampaio. Automatic test-case generation. In Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Testing Techniques in Software Engineering*, volume 6153 of *Lecture Notes in Computer Science*, pages 59–103. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[94] Patrícia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards property oriented testing. *Electronic Notes in Theoretical Computer Science*, 184:3–19, 2007.

[95] John D. McGregor and David A. Sykes. *A practical guide to testing object-oriented software*. Addison-Wesley, Boston, MA, USA, 2001.

[96] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. Formal testing of systems presenting soft and hard deadlines. In *FSEN'07: Proc. of the Int. Symp. on Fundamentals of Software Engineering*, volume 4767 of *LNCS*, pages 160–174. Springer, 2007.

[97] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. Extending EFSMs to specify and test timed systems with action durations and time-outs. *IEEE Trans. Comput.*, 57(6):835–844, 2008.

[98] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. Formal testing from timed finite state machines. *Comput. Netw.*, 52(2):432–460, 2008.

[99] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. A formal framework to test soft and hard deadlines in timed systems. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2011.

[100] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 396–397, Washington, DC, USA, 2004. IEEE Computer Society.

[101] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2nd edition, 2004.

[102] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from CSP models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 258–273, Berlin, Heidelberg, 2008. Springer-Verlag.

[103] Manuel Núñez and Ismael Rodríguez. Conformance testing relations for timed systems. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Software Testing*, volume 3997 of *Lecture Notes in Computer Science*, pages 103–117. Springer Berlin / Heidelberg, 2006.

[104] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11:339–353, October 2009.

[105] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2007.

[106] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., 7th edition, 2009.

[107] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988.

[108] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In *IFM'00: Proc. of the Second Int. Conf. on Integrated Formal Methods*, pages 338–357. Springer, 2000.

[109] Steve Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[110] Ian Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, Boston, MA, USA, 9th edition, 2010.

[111] Mitsuo Takaki, Diego Cavalcanti, Rohit Gheyi, Juliano Iyoda, Marcelo D'Amorim, and Ricardo B. Prudêncio. Randomized constraint solvers: a comparative study. *Innov. Syst. Softw. Eng.*, 6:243–253, September 2010.

[112] The FreeRTOS.org Project. FreeRTOS. http://www.freertos.org.

[113] Omer Nguena Timo, Hervé Marchand, and Antoine Rollet. Automatic test generation for data-flow reactive systems with time constraints. In *ICTSS'10: Proceedings of the 22nd IFIP International Conference on Testing Software and Systems*, pages 25–30, 2010. Short Paper.

[114] Omer Nguena Timo and Antoine Rollet. Conformance testing of variable driven automata. In *WFCS'10: Proceedings of the 8th IEEE International Workshop on Factory Communication Systems*, pages 241–248. IEEE Computer Society, 2010.

[115] Dante Torres, Daniel Leitao, and Flávia Barros. Motorola SpecNL: A hybrid system to generate nl descriptions from test case specifications. In *HIS '06: Proceedings of the*

Sixth International Conference on Hybrid Intelligent Systems, page 45, Washington, DC, USA, 2006. IEEE Computer Society.

[116] Jan Tretmans. Conformance testing with labelled transition systems: implementation relations and test generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, 1996.

[117] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In *TACAS'96: Proc. of the Second Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 127–146. Springer, 1996.

[118] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99: Proc. of the 10th Int. Conf. on Concurrency Theory*, pages 46–65. Springer, 1999.

[119] Jan Tretmans and Ed Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proceedings of the First European Conference on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, 2003.

[120] Sabrina von Styp, Henrik Bohnenkamp, and Julien Schmaltz. A conformance testing relation for symbolic timed automata. In Krishnendu Chatterjee and Thomas Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *Lecture Notes in Computer Science*, pages 243–255. Springer Berlin / Heidelberg, 2010.

[121] Rob Williams. *Real-Time Systems Development.* Butterworth-Heinemann, Oxford, UK, 2006.

[122] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. *Form. Methods Syst. Des.*, 11:113–136, August 1997.

[123] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 243–258, London, UK, UK, 1995. Chapman & Hall, Ltd.

[124] Mao Zheng, Vasu Alagar, and Olga Ormandjieva. Automated generation of test suites from formal specifications of real-time reactive systems. *J. Syst. Softw.*, 81(2):286–304, 2008.

Apêndice A

Proofs

Proof of Theorem 3.1.

Proof of soundness:

According to Definition 3.7 a test suite is sound if all of its test cases are sound. Furthermore, a test case TC is sound for S and **conf** if $\forall$SUT,

$$\text{SUT } \textbf{conf } S \Rightarrow \neg(TC \text{ may reject SUT}).$$

Using the contraposition principle, we need to prove that if a test case TC may reject a SUT (implementing the specification S), then $\neg$(SUT **conf** S). Thus, we need to prove that $\forall$SUT,

$$TC \text{ may reject SUT} \Rightarrow \neg(\text{SUT } \textbf{conf } S).$$

By Definition 3.6 we need to prove that $\forall$SUT,

$$\exists \sigma \in \text{Traces}(TC \mid\mid \text{SUT}) : \text{verdict}(\sigma) = \textit{Fail} \Rightarrow \neg(\text{SUT } \textbf{conf } S).$$

Let SUT be an arbitrary implementation such that $\exists \sigma \in \text{Traces}(TC \mid\mid \text{SUT}) : \text{verdict}(\sigma) = \textit{Fail}$. Then, let $\sigma = [a_1]\omega_1\ [a_2]\omega_2\ \ldots\ [a_n]\omega_n \in ([A]L)^*$ be the trace corresponding to the interaction between TC and SUT until the verdict *Fail* is emitted. Also, let $\sigma_{n-1} = [a_1]\omega_1\ [a_2]\omega_2\ \ldots\ [a_{n-1}]\omega_{n-1}$ be a trace excluding $[a_n]\omega_n$. According to the verdicts definition (page 16), if a *Fail* is emitted then $[a_n]\omega_n$ is an output action. Thus, $Out(\text{SUT } \textit{after } \sigma_{n-1}) \neq \emptyset$ because $[a_n]\omega_n \in Out(\text{SUT } \textit{after } \sigma_{n-1})$.

Since *Fail* is obtained, $[a_n]\omega_n \notin Out(S \textit{ after } \sigma_{n-1})$. Hence, $Out(\text{SUT } \textit{after } \sigma_{n-1}) \not\subseteq Out(S \textit{ after } \sigma_{n-1})$ and consequently $\neg$(SUT **conf** S).

Then, $\forall$SUT, TC may reject SUT $\Rightarrow \neg$(SUT **conf** S) and, consequently,

$$\forall TC\ \forall \text{SUT},\ TC \text{ may reject SUT} \Rightarrow \neg(\text{SUT}\ \textbf{conf}\ S).$$

Proof of exhaustiveness:

For proving that the test suites generated by LTS-BT are exhaustive, we need to prove that for every non-conforming SUT there is a test purpose TP and a way of generating a test case TC from S and TP, such that TC may reject SUT.

According to Definition 3.7 a test suite is exhaustive for S and **conf** if $\forall$SUT,

$$\neg(\text{SUT}\ \textbf{conf}\ S) \Rightarrow \exists TC : TC \text{ may reject SUT}.$$

By Definition 3.3, if $\neg$(SUT **conf** S) then there is a trace $\sigma = [a_1]\omega_1\ [a_2]\omega_2\ \ldots\ [a_{n-1}]\omega_{n-1}\ [a_n]\omega_n \in \mathit{Traces}(S)$ and an output action $[a_{n+1}]\omega_{n+1} \in [A \setminus \{steps, conditions, beginInterruption\_X\}]L_O$ such that

$$[a_{n+1}]\omega_{n+1} \in Out(\text{SUT}\ \textit{after}\ \sigma) \text{ and } [a_{n+1}]\omega_{n+1} \notin Out(S\ \textit{after}\ \sigma).$$

Let $[a'_{n+1}]\omega'_{n+1}$ be the correct output action such that $[a'_{n+1}]\omega'_{n+1} \in Out(S\ \textit{after}\ \sigma)$. Thus, σ and $[a'_{n+1}]\omega'_{n+1}$ can be used to define the following TP:

$$\text{“}\omega_1\ \omega_2\ \ldots\ \omega_{n-1}\ \omega_n; \omega'_{n+1}; \text{Accept”}.$$

Finally, a test case TC is generated based on S and the defined TP. Thus, during the test case execution the SUT produces $[a_{n+1}]\omega_{n+1}$ instead of $[a'_{n+1}]\omega'_{n+1}$. In this case, a *Fail* verdict is emitted as expected. Hence, TC may reject SUT according to Definition 3.6. □

Proof of Theorem 6.1.

Proof of soundness:

Let $[\![TC]\!] = \langle S, S^0, Act, T \rangle$ be the TIOLTS semantics of the test case $TC = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$. According to Definition 6.9 a test suite is sound if all of its test cases are sound. Furthermore, a test case TC is sound for $\mathcal{S}$ and **tioco** if $\forall \mathcal{I}$,

$$\mathcal{I}\ \textbf{tioco}\ \mathcal{S} \Rightarrow \neg(TC\ \textit{may reject}\ \mathcal{I}).$$

Using the contraposition principle, we need to prove that if a test case TC may reject $\mathcal{I}$ (implementing the specification $\mathcal{S}$), then $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$. Thus, we need to prove that $\forall \mathcal{I}$,

$$TC\ \textit{may reject}\ \mathcal{I} \Rightarrow \neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}).$$

By Definition 6.8 we need to prove that $\forall \mathcal{I}$,

$$\exists \sigma \in \textit{Traces}([\![TC]\!]\ ||\ \textit{ObservableTraces}(\mathcal{I})) : \textit{verdict}(\sigma) = \textbf{Fail} \Rightarrow \neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}).$$

Let $\mathcal{I}$ be an arbitrary implementation such that $\exists \sigma \in \textit{Traces}([\![TC]\!]\ ||\ \textit{ObservableTraces}(\mathcal{I})) : \textit{verdict}(\sigma) = \textbf{Fail}$. Then, let $\sigma = a_1\ a_2\ \ldots\ a_n \in (Act \backslash \Lambda^{\tau})^*$ be the trace corresponding to the interaction between $[\![TC]\!]$ and an observable behaviour of $\mathcal{I}$ until the verdict **Fail** is emitted. Also, let $\sigma_{n-1} = a_1\ a_2\ \ldots\ a_{n-1}$ be a trace excluding a_n. According to the verdicts definition (Section 6.3), if a **Fail** is emitted then a_n is either an output action or a time-elapsing action. Thus, $Out(\mathcal{I}\ \textit{after}\ \sigma_{n-1}) \neq \emptyset$ because $a_n \in Out(\mathcal{I}\ \textit{after}\ \sigma_{n-1})$.

Since **Fail** is obtained, $a_n \notin Out(\mathcal{S}\ \textit{after}\ \sigma_{n-1})$. Hence, $Out(\mathcal{I}\ \textit{after}\ \sigma_{n-1}) \not\subseteq Out(\mathcal{S}\ \textit{after}\ \sigma_{n-1})$ and consequently $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$.

Then, $\forall \mathcal{I}$, TC *may reject* $\mathcal{I} \Rightarrow \neg(\mathcal{I}$ **tioco** $\mathcal{S})$ and, consequently,

$$\forall TC\ \forall \mathcal{I},\ TC\ \textit{may reject}\ \mathcal{I} \Rightarrow \neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}).$$

Proof of exhaustiveness:

For proving that the test suites generated by our approach are exhaustive, we need to prove that for every non-conforming $\mathcal{I}$ there is a test purpose TP and a way of generating a test case TC from $\mathcal{S}$ and TP, such that TC may reject $\mathcal{I}$.

According to Definition 6.9 a test suite is exhaustive for $\mathcal{S}$ and **tioco** if $\forall \mathcal{I}$,

$$\neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}) \Rightarrow \exists TC : TC\ \textit{may reject}\ \mathcal{I}.$$

By Definition 6.2, if $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$ then there is a trace $\sigma = a_1\ a_2\ \ldots\ a_{n-1}\ a_n \in$ *ObservableTraces*$(\mathcal{S})$ and an output event a_{n+1} (i.e., an output action or time-elapsing action) such that

$$a_{n+1} \in Out(\mathcal{I}\ \textit{after}\ \sigma) \text{ and } a_{n+1} \notin Out(\mathcal{S}\ \textit{after}\ \sigma).$$

Let a'_{n+1} be the correct output event such that $a'_{n+1} \in Out(\mathcal{S}\ \textit{after}\ \sigma)$. Thus, σ and a'_{n+1} can be used to define a TP with the path "$a_1\ a_2\ \ldots\ a_{n-1}\ a_n\ a'_{n+1}$" leading to an *Accept* location.

Finally, a test case TC is generated based on $\mathcal{S}$ and the defined TP. Thus, during the test case execution $\mathcal{I}$ produces a_{n+1} instead of a'_{n+1}. In this case, a **Fail** verdict is emitted as expected. Hence, TC may reject $\mathcal{I}$ according to Definition 6.8. □

Apêndice B

TIOSTS Models

This appendix presents all TIOSTS models automatically generated by the prototype tool during the execution of the case studies described in Chapter 8.

B.1 TIOSTS Models of the Burglar Alarm System Case Study

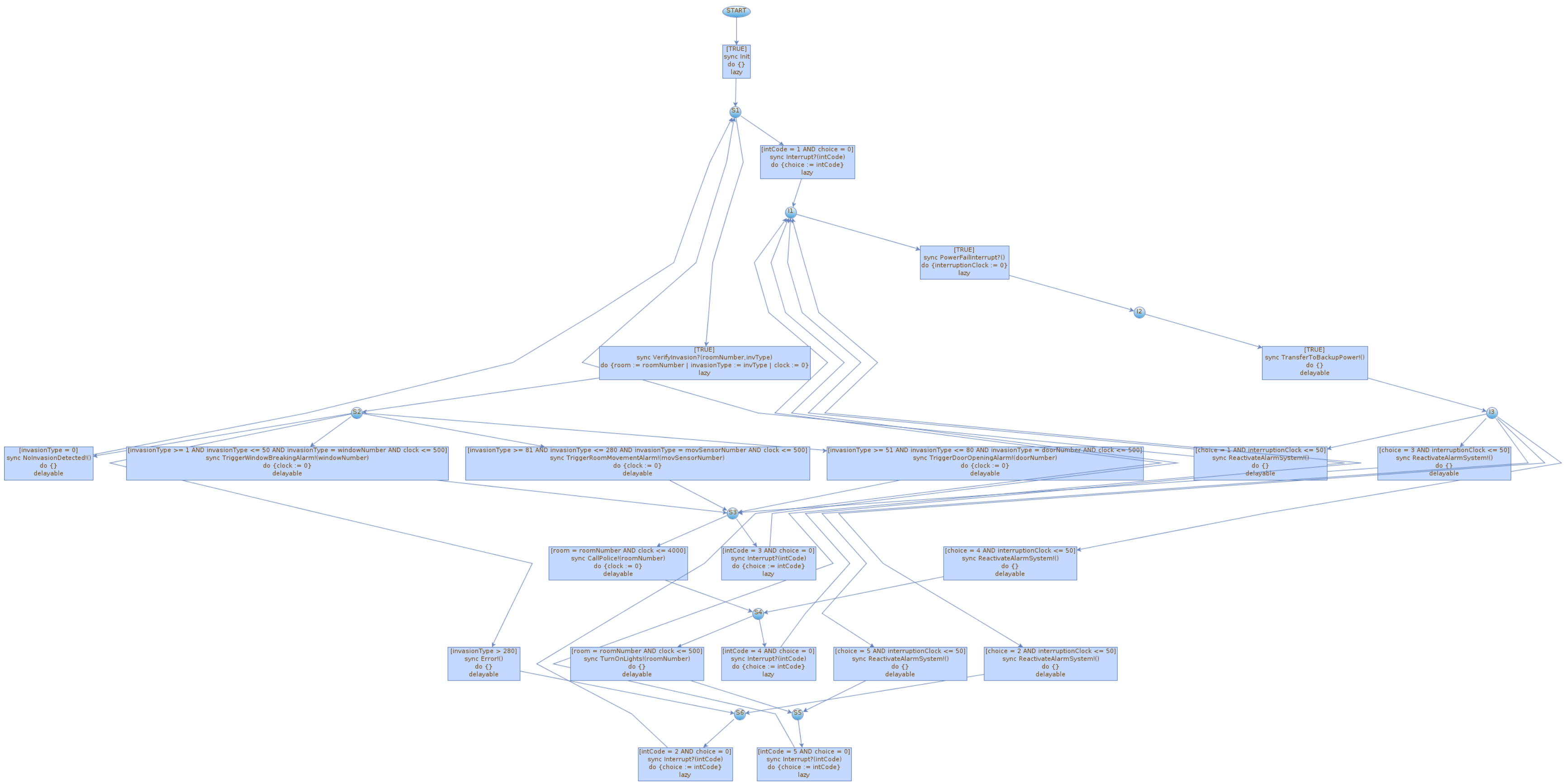

START
[TRUE]
sync Init
do {}
lazy
S1
[intCode = 1 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
I1
[TRUE]
sync PowerFailInterrupt?()
do {interruptionClock := 0}
lazy
I2
[TRUE]
sync TransferToBackupPower!()
do {}
delayable
I3
[TRUE]
sync VerifyInvasion?(roomNumber,invType)
do {room := roomNumber | invasionType := invType | clock := 0}
lazy
S2
[invasionType = 0]
sync NoInvasionDetected!()
do {}
delayable
[invasionType >= 1 AND invasionType <= 50 AND invasionType = windowNumber AND clock <= 500]
sync TriggerWindowBreakingAlarm!(windowNumber)
do {clock := 0}
delayable
[invasionType >= 81 AND invasionType <= 280 AND invasionType = movSensorNumber AND clock <= 500]
sync TriggerRoomMovementAlarm!(movSensorNumber)
do {clock := 0}
delayable
[invasionType >= 51 AND invasionType <= 80 AND invasionType = doorNumber AND clock <= 500]
sync TriggerDoorOpeningAlarm!(doorNumber)
do {clock := 0}
delayable
[choice = 1 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
[choice = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
S3
[room = roomNumber AND clock <= 4000]
sync CallPolice!(roomNumber)
do {clock := 0}
delayable
[intCode = 3 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
[choice = 4 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
S4
[invasionType > 280]
sync Error!()
do {}
delayable
[room = roomNumber AND clock <= 500]
sync TurnOnLights!(roomNumber)
do {}
delayable
[intCode = 4 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
[choice = 5 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
[choice = 2 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
S6
S5
[intCode = 2 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
[intCode = 5 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy

Figura B.1: Burglar Alarm System Specification

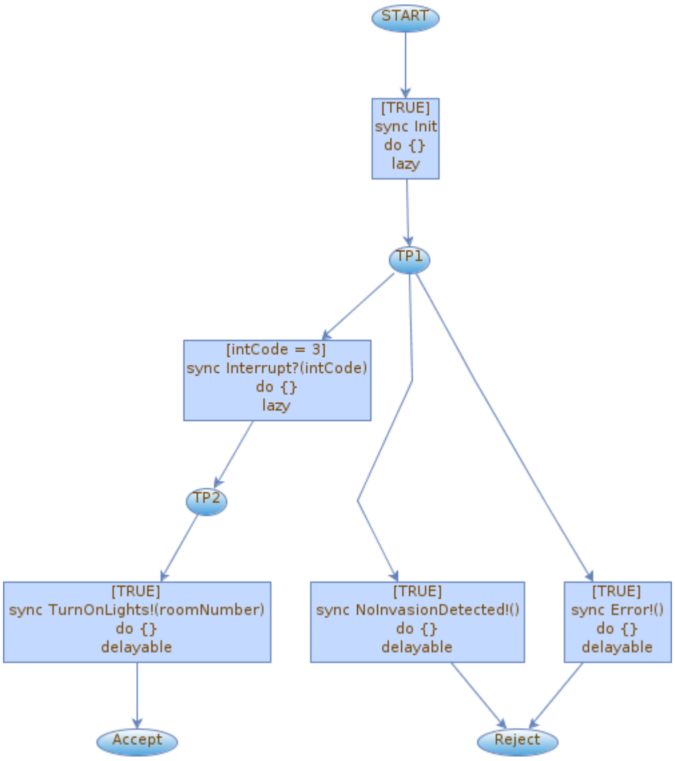


Figura B.2: Test Purpose

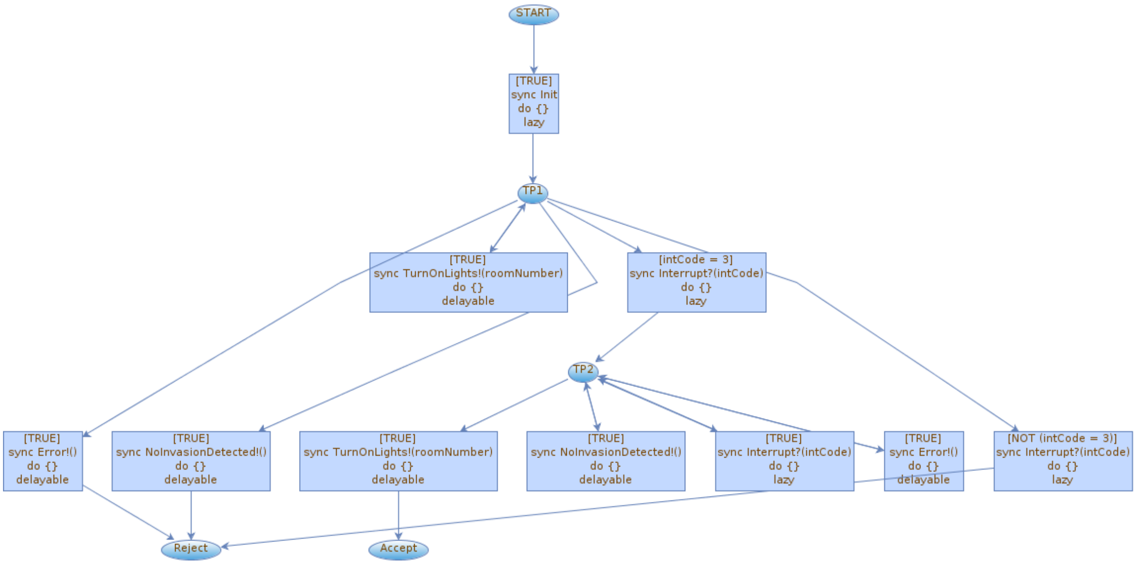


Figura B.3: Completed Test Purpose

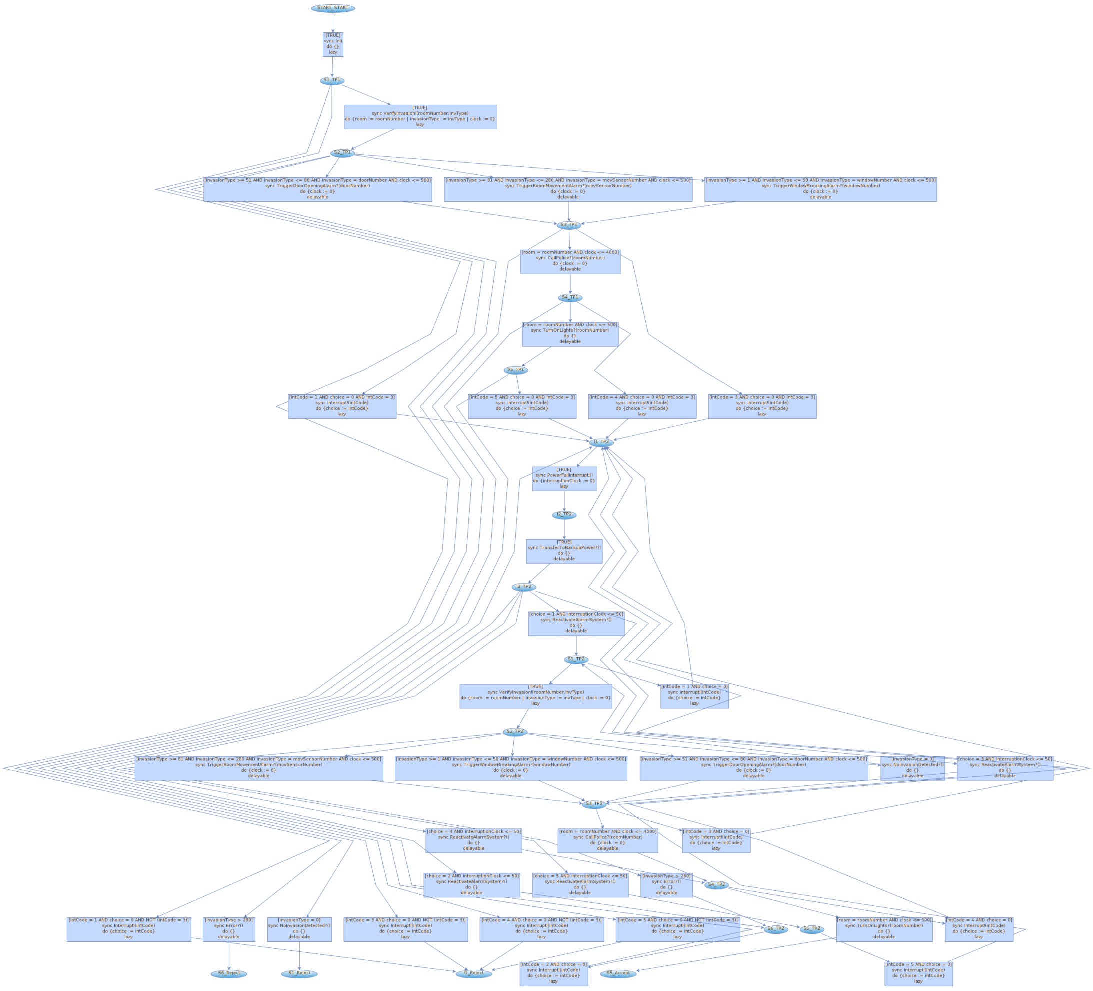

START_START
[TRUE] sync Init do {} lazy
S1_TP1
[TRUE] sync VerifyInvasion!(roomNumber,invType) do {room := roomNumber | invasionType := invType | clock := 0} lazy
S2_TP1
[invasionType >= 51 AND invasionType <= 80 AND invasionType = doorNumber AND clock <= 500] sync TriggerDoorOpeningAlarm?(doorNumber) do {clock := 0} delayable
[invasionType >= 81 AND invasionType <= 280 AND invasionType = movSensorNumber AND clock <= 500] sync TriggerRoomMovementAlarm?(movSensorNumber) do {clock := 0} delayable
[invasionType >= 1 AND invasionType <= 50 AND invasionType = windowNumber AND clock <= 500] sync TriggerWindowBreakingAlarm?(windowNumber) do {clock := 0} delayable
S3_TP1
[room = roomNumber AND clock <= 4000] sync CallPolice?(roomNumber) do {clock := 0} delayable
S4_TP1
[room = roomNumber AND clock <= 500] sync TurnOnLights?(roomNumber) do {} delayable
S5_TP1
[intCode = 1 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 5 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 4 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 3 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
I1_TP2
[TRUE] sync PowerFailInterrupt!() do {interruptionClock := 0} lazy
I2_TP2
[TRUE] sync TransferToBackupPower?() do {} delayable
I3_TP2
[choice = 1 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
S1_TP2
[TRUE] sync VerifyInvasion!(roomNumber,invType) do {room := roomNumber | invasionType := invType | clock := 0} lazy
[intCode = 1 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
S2_TP2
[invasionType >= 81 AND invasionType <= 280 AND invasionType = movSensorNumber AND clock <= 500] sync TriggerRoomMovementAlarm?(movSensorNumber) do {clock := 0} delayable
[invasionType >= 1 AND invasionType <= 50 AND invasionType = windowNumber AND clock <= 500] sync TriggerWindowBreakingAlarm?(windowNumber) do {clock := 0} delayable
[invasionType >= 51 AND invasionType <= 80 AND invasionType = doorNumber AND clock <= 500] sync TriggerDoorOpeningAlarm?(doorNumber) do {clock := 0} delayable
[invasionType = 0] sync NoInvasionDetected?() do {} delayable
[choice = 3 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
S3_TP2
[choice = 4 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
[room = roomNumber AND clock <= 4000] sync CallPolice?(roomNumber) do {clock := 0} delayable
[intCode = 3 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
[choice = 2 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
[choice = 5 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
[invasionType > 280] sync Error?() do {} delayable
S4_TP2
[intCode = 1 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
[invasionType > 280] sync Error?() do {} delayable
[invasionType = 0] sync NoInvasionDetected?() do {} delayable
[intCode = 3 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 4 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 5 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
S6_TP2
S5_TP2
[room = roomNumber AND clock <= 500] sync TurnOnLights?(roomNumber) do {} delayable
[intCode = 4 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
S6_Reject
S1_Reject
I1_Reject
[intCode = 2 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
S5_Accept
[intCode = 5 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy

Figura B.4: Synchronous Product

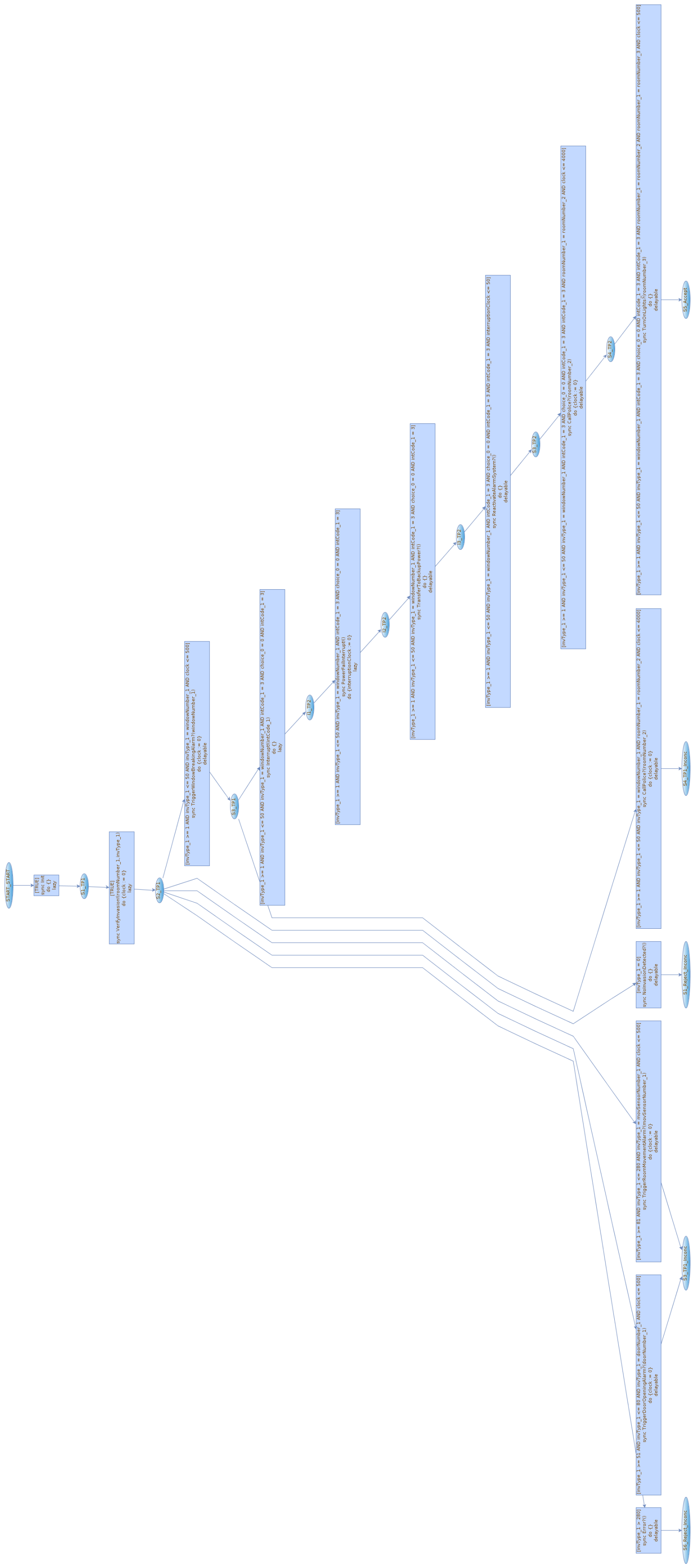

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync VerifyInvasion!(roomNumber_1,invType_1)
do {clock := 0}
lazy
S2_TP1
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND clock <= 500]
sync TriggerWindowBreakingAlarm?(windowNumber_1)
do {clock := 0}
delayable
S3_TP1
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync Interrupt!(intCode_1)
do {}
lazy
I1_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync PowerFailInterrupt!()
do {interruptionClock := 0}
lazy
I2_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync TransferToBackupPower?()
do {}
delayable
I3_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem?()
do {}
delayable
S3_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND roomNumber_1 = roomNumber_3 AND clock <= 500]
sync TurnOnLights?(roomNumber_3)
do {}
delayable
S5_Accept
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP1_Inconc
[invType_1 = 0]
sync NoInvasionDetected?()
do {}
delayable
S1_Reject_Inconc
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND clock <= 500]
sync TriggerRoomMovementAlarm?(movSensorNumber_1)
do {clock := 0}
delayable
S3_TP1_Inconc
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND clock <= 500]
sync TriggerDoorOpeningAlarm?(doorNumber_1)
do {clock := 0}
delayable
[invType_1 > 280]
sync Error?()
do {}
delayable
S6_Reject_Inconc

Figura B.5: Test Case 01

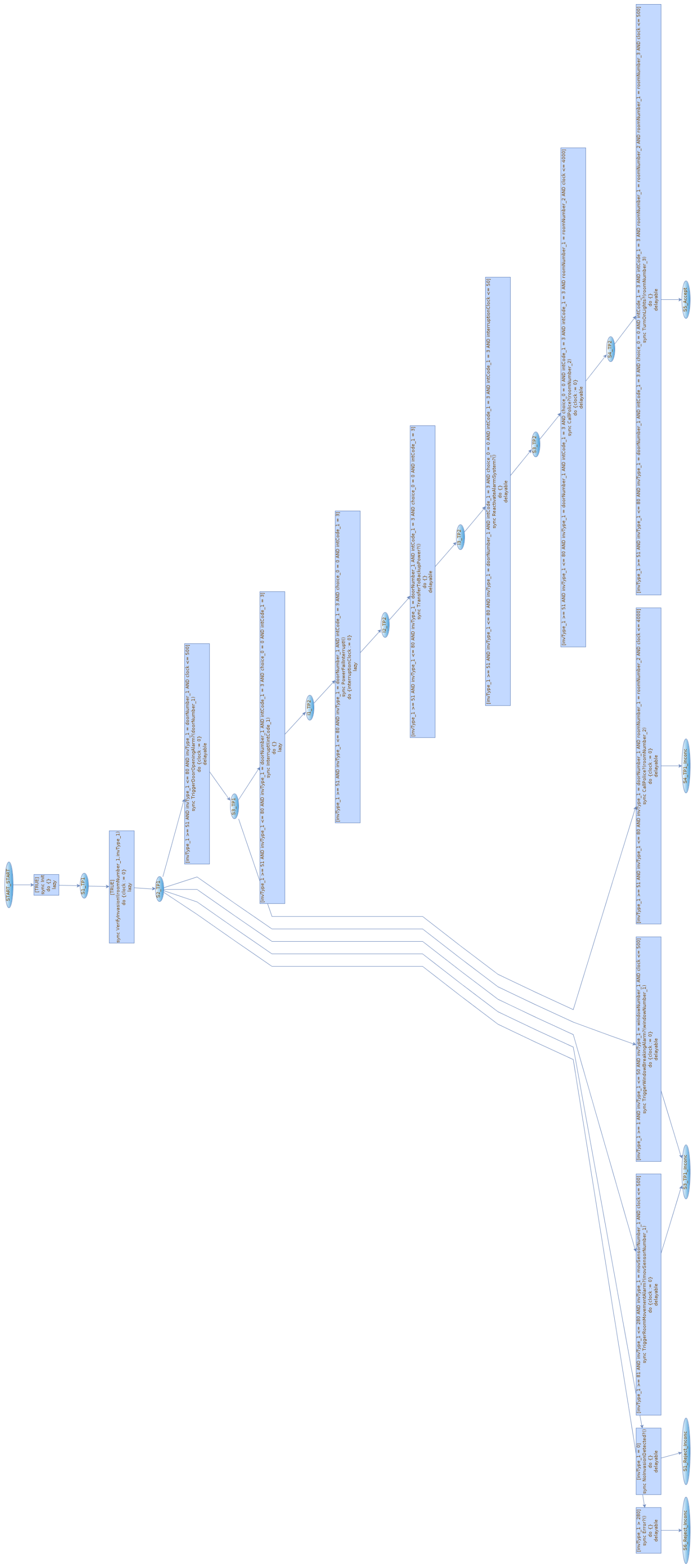

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync VerifyInvasion!(roomNumber_1,invType_1)
do {clock := 0}
lazy
S2_TP1
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND clock <= 500]
sync TriggerDoorOpeningAlarm?(doorNumber_1)
do {clock := 0}
delayable
S3_TP1
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync Interrupt!(intCode_1)
do {}
lazy
I1_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync PowerFailInterrupt!()
do {interruptionClock := 0}
lazy
I2_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync TransferToBackupPower?()
do {}
delayable
I3_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem?()
do {}
delayable
S3_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND roomNumber_1 = roomNumber_3 AND clock <= 500]
sync TurnOnLights?(roomNumber_3)
do {}
delayable
S5_Accept
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP1_Inconc
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND clock <= 500]
sync TriggerWindowBreakingAlarm?(windowNumber_1)
do {clock := 0}
delayable
S3_TP1_Inconc
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND clock <= 500]
sync TriggerRoomMovementAlarm?(movSensorNumber_1)
do {clock := 0}
delayable
[invType_1 = 0]
sync NoInvasionDetected?()
do {}
delayable
S1_Reject_Inconc
[invType_1 > 280]
sync Error?()
do {}
delayable
S6_Reject_Inconc

Figura B.6: Test Case 02

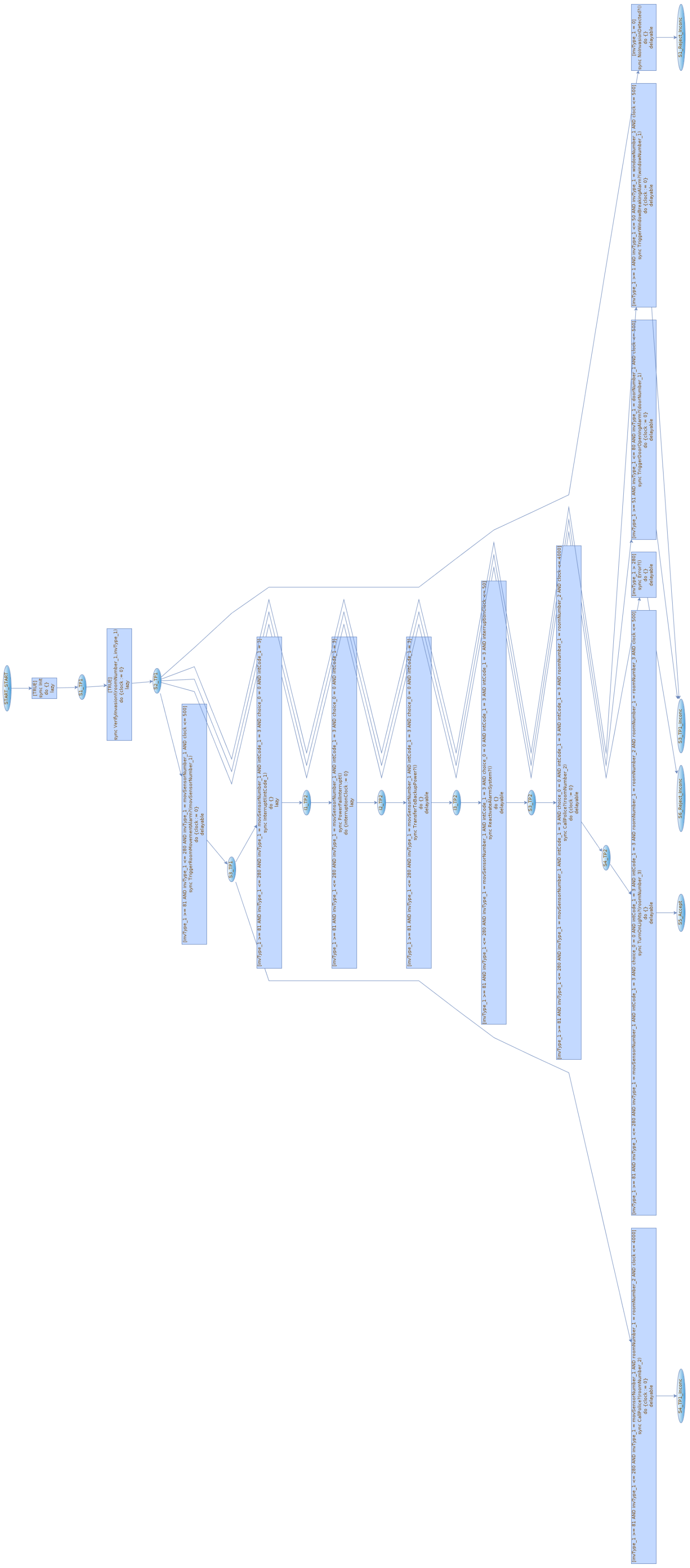

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync VerifyInvasion!(roomNumber_1,invType_1)
do {clock := 0}
lazy
S2_TP1
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND clock <= 500]
sync TriggerRoomMovementAlarm?(movSensorNumber_1)
do {clock := 0}
delayable
S3_TP1
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync Interrupt!(intCode_1)
do {}
lazy
I1_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync PowerFailInterrupt!()
do {interruptionClock := 0}
lazy
I2_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync TransferToBackupPower?()
do {}
delayable
I3_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem?()
do {}
delayable
S3_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND roomNumber_1 = roomNumber_3 AND clock <= 500]
sync TurnOnLights?(roomNumber_3)
do {}
delayable
S5_Accept
S6_Reject_Inconc
S3_TP1_Inconc
[invType_1 > 280]
sync Error?()
do {}
delayable
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND clock <= 500]
sync TriggerDoorOpeningAlarm?(doorNumber_1)
do {clock := 0}
delayable
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND clock <= 500]
sync TriggerWindowBreakingAlarm?(windowNumber_1)
do {clock := 0}
delayable
[invType_1 = 0]
sync NoInvasionDetected?()
do {}
delayable
S1_Reject_Inconc
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP1_Inconc

Figura B.7: Test Case 03

B.2 TIOSTS Models of the Automatic Guided Vehicle System Case Study

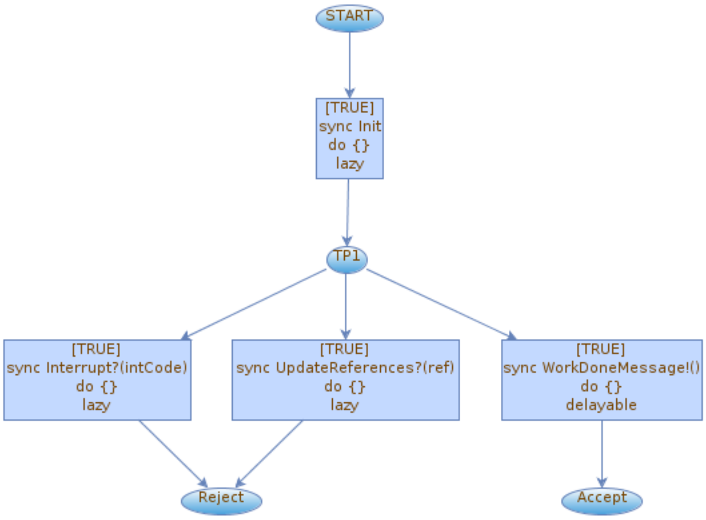


Figura B.8: Test Purpose TP1

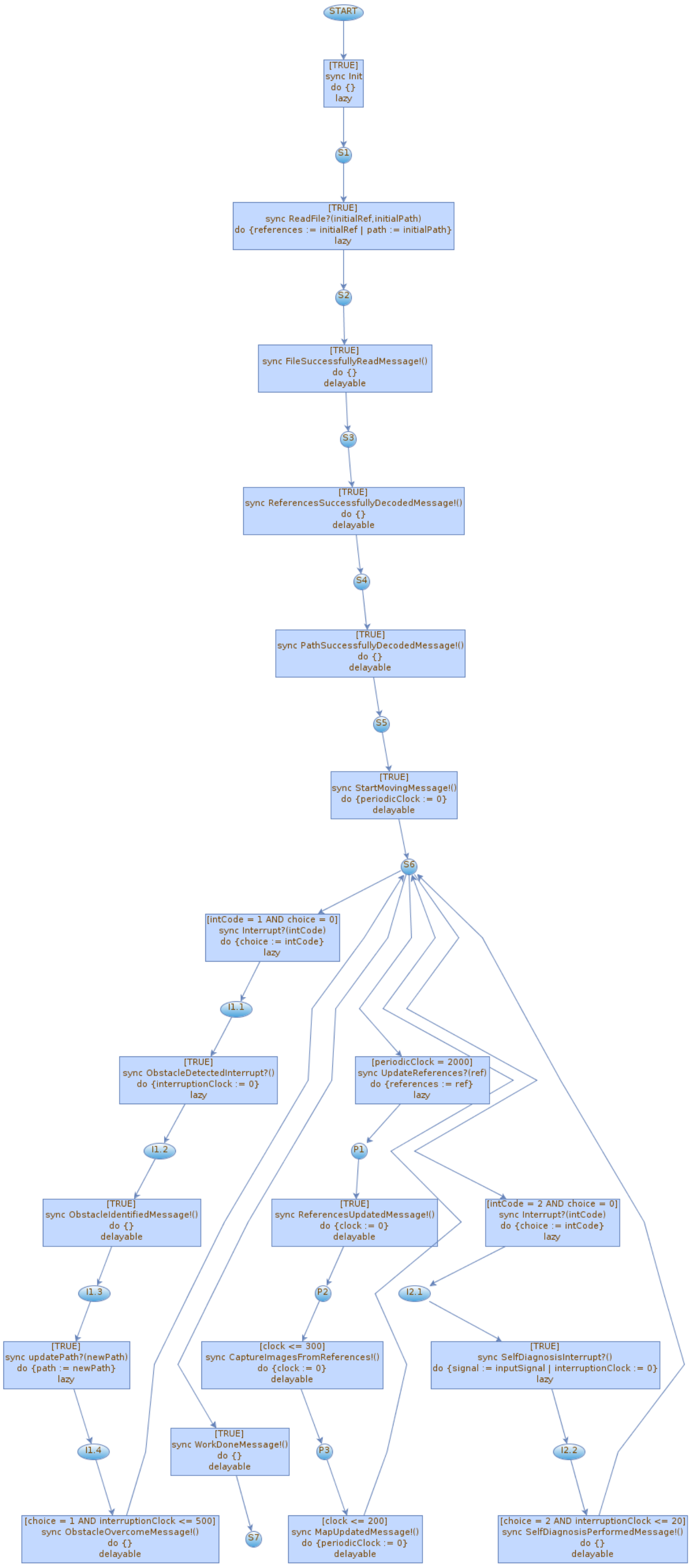

START
[TRUE]
sync Init
do {}
lazy
S1
[TRUE]
sync ReadFile?(initialRef,initialPath)
do {references := initialRef | path := initialPath}
lazy
S2
[TRUE]
sync FileSuccessfullyReadMessage!()
do {}
delayable
S3
[TRUE]
sync ReferencesSuccessfullyDecodedMessage!()
do {}
delayable
S4
[TRUE]
sync PathSuccessfullyDecodedMessage!()
do {}
delayable
S5
[TRUE]
sync StartMovingMessage!()
do {periodicClock := 0}
delayable
S6
[intCode = 1 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
I1.1
[TRUE]
sync ObstacleDetectedInterrupt?()
do {interruptionClock := 0}
lazy
I1.2
[TRUE]
sync ObstacleIdentifiedMessage!()
do {}
delayable
I1.3
[TRUE]
sync updatePath?(newPath)
do {path := newPath}
lazy
I1.4
[choice = 1 AND interruptionClock <= 500]
sync ObstacleOvercomeMessage!()
do {}
delayable
[TRUE]
sync WorkDoneMessage!()
do {}
delayable
S7
[periodicClock = 2000]
sync UpdateReferences?(ref)
do {references := ref}
lazy
P1
[TRUE]
sync ReferencesUpdatedMessage!()
do {clock := 0}
delayable
P2
[clock <= 300]
sync CaptureImagesFromReferences!()
do {clock := 0}
delayable
P3
[clock <= 200]
sync MapUpdatedMessage!()
do {periodicClock := 0}
delayable
[intCode = 2 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
I2.1
[TRUE]
sync SelfDiagnosisInterrupt?()
do {signal := inputSignal | interruptionClock := 0}
lazy
I2.2
[choice = 2 AND interruptionClock <= 20]
sync SelfDiagnosisPerformedMessage!()
do {}
delayable

Figura B.9: Automatic Guided Vehicle System Specification

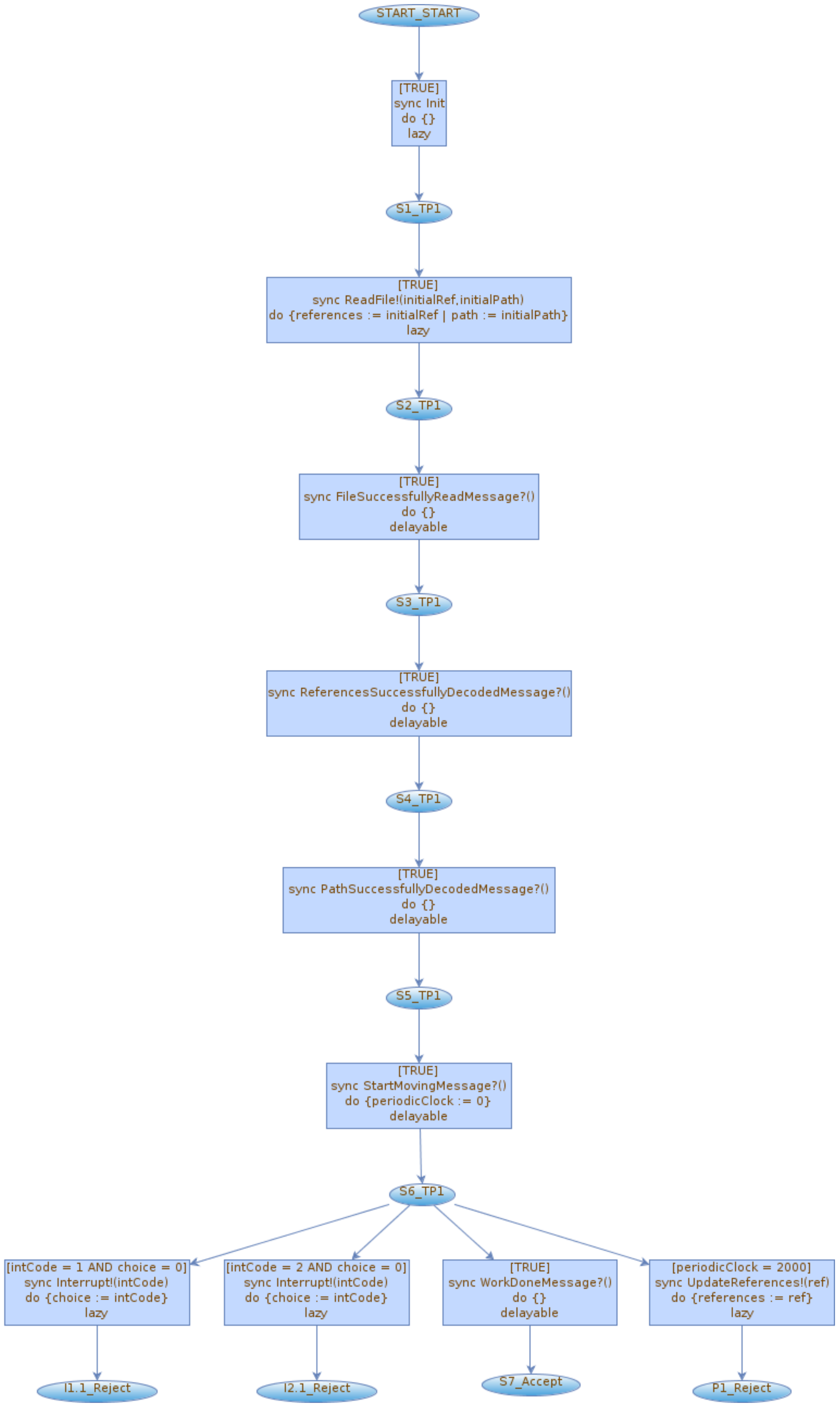


Figura B.10: Synchronous Product between AVG System Specification and Test Purpose TP1

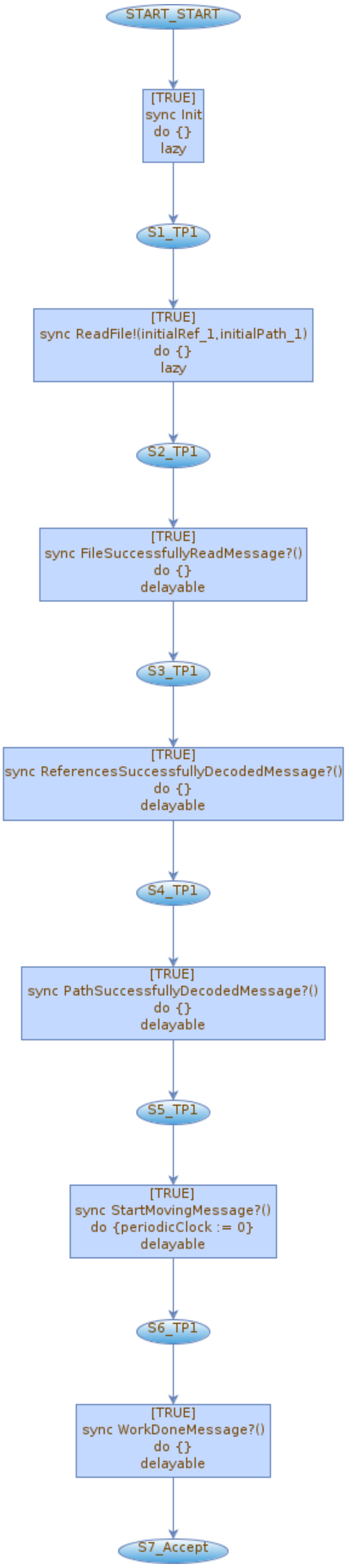


Figura B.11: Test Case of the First Scenario

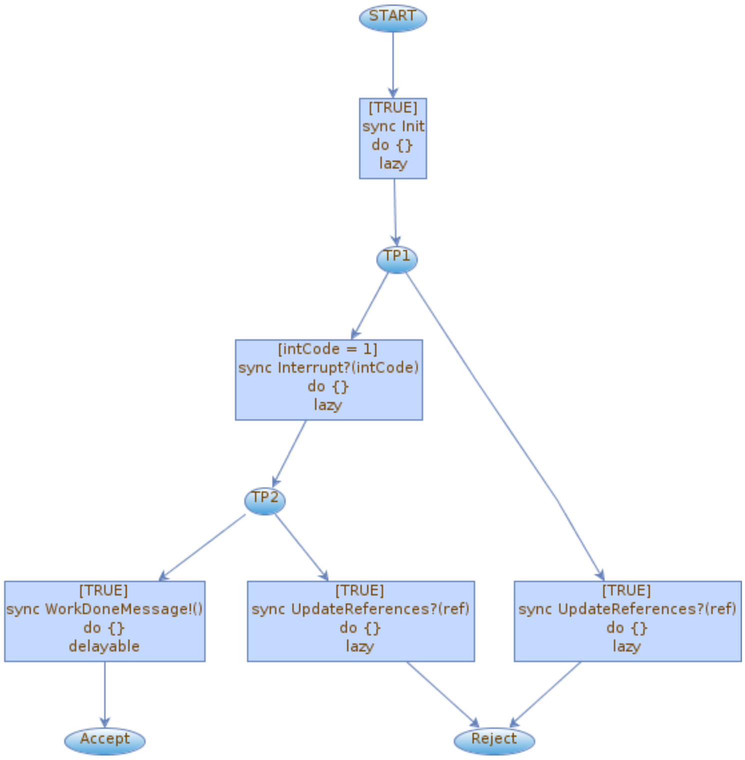


Figura B.12: Test Purpose TP2

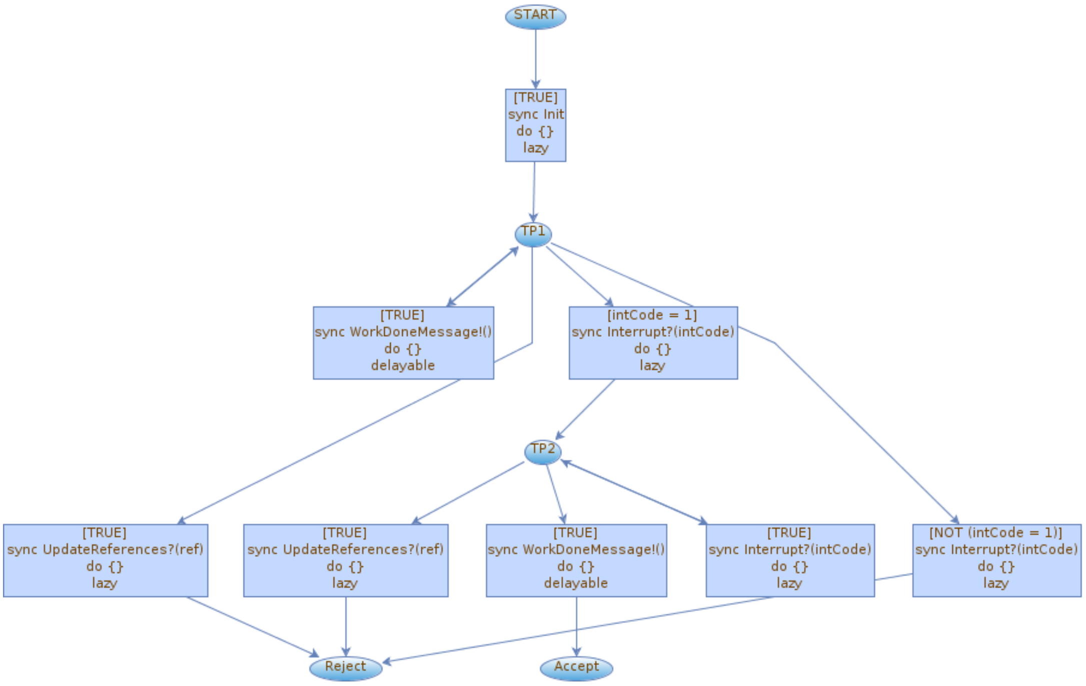


Figura B.13: Completed Test Purpose TP2

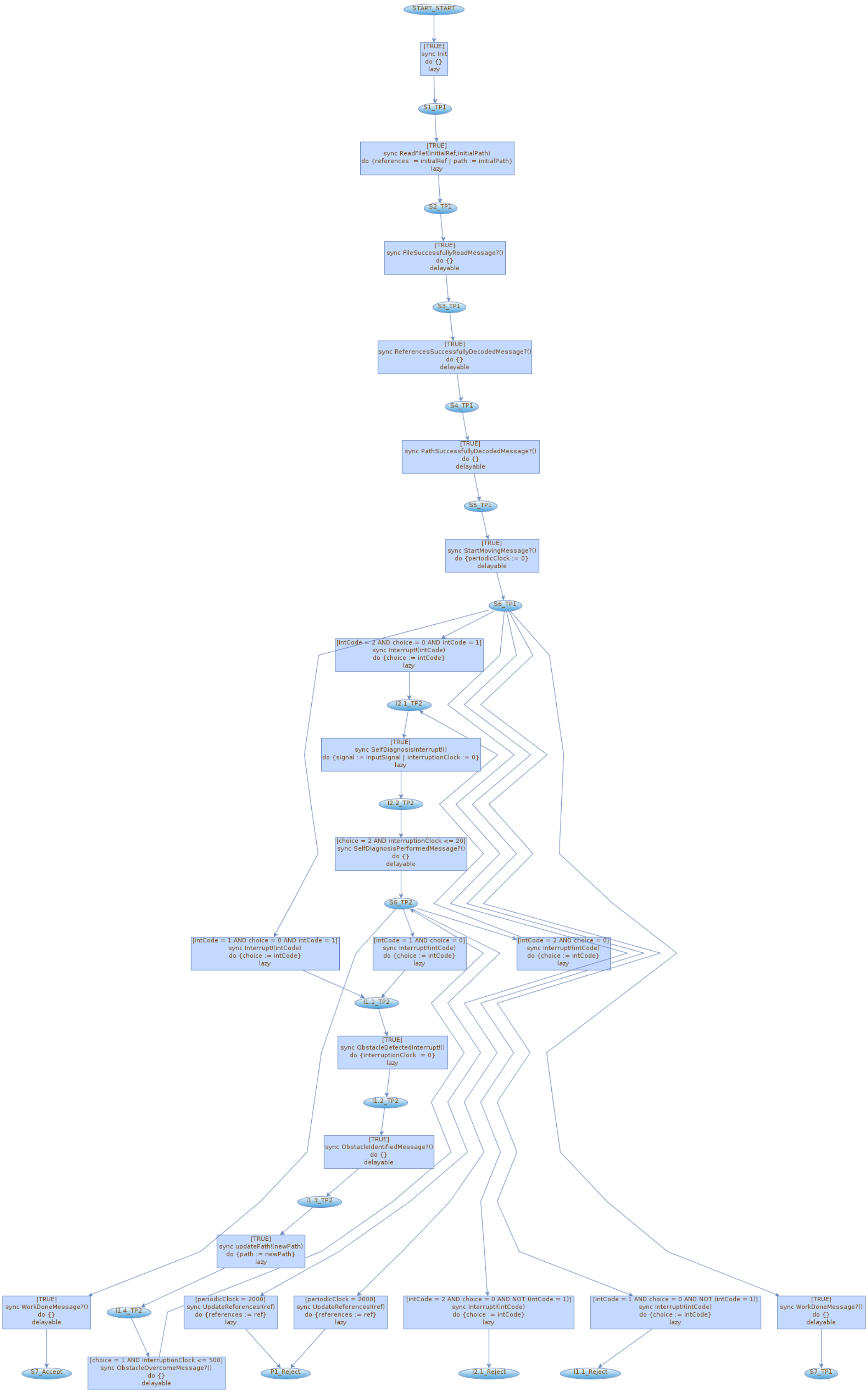

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync ReadFile!(initialRef,initialPath)
do {references := initialRef | path := initialPath}
lazy
S2_TP1
[TRUE]
sync FileSuccessfullyReadMessage?()
do {}
delayable
S3_TP1
[TRUE]
sync ReferencesSuccessfullyDecodedMessage?()
do {}
delayable
S4_TP1
[TRUE]
sync PathSuccessfullyDecodedMessage?()
do {}
delayable
S5_TP1
[TRUE]
sync StartMovingMessage?()
do {periodicClock := 0}
delayable
S6_TP1
[intCode = 2 AND choice = 0 AND intCode = 1]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
I2.1_TP2
[TRUE]
sync SelfDiagnosisInterrupt!()
do {signal := inputSignal | interruptionClock := 0}
lazy
I2.2_TP2
[choice = 2 AND interruptionClock <= 20]
sync SelfDiagnosisPerformedMessage?()
do {}
delayable
S6_TP2
[intCode = 1 AND choice = 0 AND intCode = 1]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[intCode = 1 AND choice = 0]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[intCode = 2 AND choice = 0]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
I1.1_TP2
[TRUE]
sync ObstacleDetectedInterrupt!()
do {interruptionClock := 0}
lazy
I1.2_TP2
[TRUE]
sync ObstacleIdentifiedMessage?()
do {}
delayable
I1.3_TP2
[TRUE]
sync updatePath!(newPath)
do {path := newPath}
lazy
[TRUE]
sync WorkDoneMessage?()
do {}
delayable
I1.4_TP2
[periodicClock = 2000]
sync UpdateReferences!(ref)
do {references := ref}
lazy
[periodicClock = 2000]
sync UpdateReferences!(ref)
do {references := ref}
lazy
[intCode = 2 AND choice = 0 AND NOT (intCode = 1)]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[intCode = 1 AND choice = 0 AND NOT (intCode = 1)]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[TRUE]
sync WorkDoneMessage?()
do {}
delayable
S7_Accept
[choice = 1 AND interruptionClock <= 500]
sync ObstacleOvercomeMessage?()
do {}
delayable
P1_Reject
I2.1_Reject
I1.1_Reject
S7_TP1

Figura B.14: Synchronous Product between AVG System Specification and Test Purpose TP2

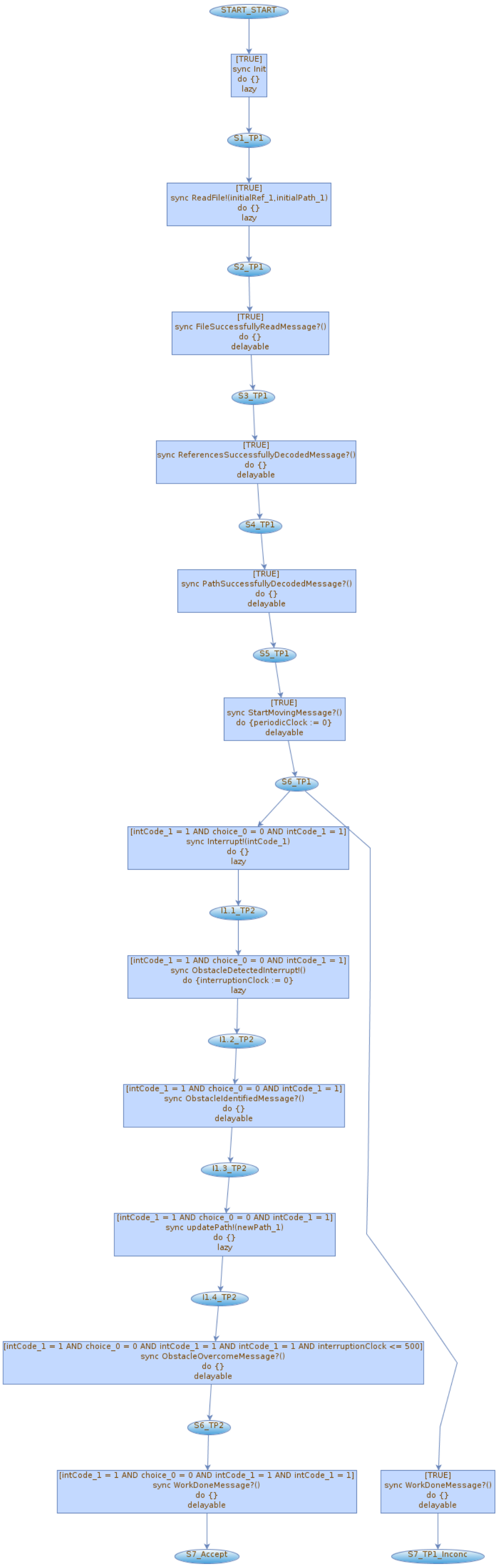

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync ReadFile!(initialRef_1,initialPath_1)
do {}
lazy
S2_TP1
[TRUE]
sync FileSuccessfullyReadMessage?()
do {}
delayable
S3_TP1
[TRUE]
sync ReferencesSuccessfullyDecodedMessage?()
do {}
delayable
S4_TP1
[TRUE]
sync PathSuccessfullyDecodedMessage?()
do {}
delayable
S5_TP1
[TRUE]
sync StartMovingMessage?()
do {periodicClock := 0}
delayable
S6_TP1
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync Interrupt!(intCode_1)
do {}
lazy
I1.1_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync ObstacleDetectedInterrupt!()
do {interruptionClock := 0}
lazy
I1.2_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync ObstacleIdentifiedMessage?()
do {}
delayable
I1.3_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync updatePath!(newPath_1)
do {}
lazy
I1.4_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1 AND intCode_1 = 1 AND interruptionClock <= 500]
sync ObstacleOvercomeMessage?()
do {}
delayable
S6_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1 AND intCode_1 = 1]
sync WorkDoneMessage?()
do {}
delayable
S7_Accept
[TRUE]
sync WorkDoneMessage?()
do {}
delayable
S7_TP1_Inconc

Figura B.15: Test Case of the Second Scenario

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Symbolic Model-Based Testing for Real-Time Systems

Wilkerson de Lucena Andrade

Thesis submitted to Coordenação do Curso de Pós-Graduação em Ciência da Computação of Universidade Federal de Campina Grande in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Research Area: Computer Science
Research Line: Software Engineering

Patrícia Duarte de Lima Machado
(Supervisor)

Campina Grande, Paraíba, Brazil

Abstract

Real-time systems are the ones whose correct behaviour depends not only on the generated results but also on whether the results are generated at the right time-points. Real-time systems are used in different contexts such as monitoring of patients in hospitals, air traffic control systems, and embedded systems in robots, appliances, vehicles, and so on. For these systems, dependability is an important property that demands rigorous application of V & V activities, since defects can mean losses in financial, environmental or human areas. As the costs and consequences of failures can be high, formal verification and model checking have been used in the V & V process. However, as these approaches have practical limitations, testing is also used as a complementary approach since it allows the execution of real scenarios within execution environments. Consequently, there is a growing interest in the search for methods, techniques and tools to support the testing of real-time systems, which poses a number of distinguishing challenges such as implementations composed of parallel activities with synchronous and asynchronous events (interruptions), with different deployment architectures, and resource limitation and time constraints on the execution environment. This thesis focuses on model-based conformance testing of real-time systems. In this context, current approaches are mostly based either on finite state machines/transition systems or on timed automata. However, most real-time systems manipulate data while being subject to time constraints. The usual solution consists in enumerating data values (in finite domains) while treating time symbolically, thus leading to the classical state explosion problem. This thesis proposes a new model of real-time systems as an extension of both symbolic transition systems and timed automata, in order to handle both data and time requirements symbolically. We propose a conformance testing theory to deal with this model and describe a test case generation process based on a combination of symbolic execution and constraint solving for the data part and symbolic analysis for timed aspects. Moreover, the proposed approach can deal with interruption testing. Finally, two case studies were performed in order to evaluate the practical application of the proposed approach.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Software systems are more and more complex and omnipresent in our lives, be they through Internet or embedded systems. In this same proportion, a special kind of application has become increasingly common, for instance, monitoring of patients in hospitals, air traffic control systems, and embedded systems in robots, mobile phones, appliances, vehicles, aircraft, and so on. All these applications have an important characteristic in common: time requirements. For such systems, the correct behaviour depends not only on the generated results but also whether the results are generated at the right time-points. Computer systems with time constraints are known as Real-Time Systems (RTS).

Most RTS are developed for specific purposes and are composed of tightly coupled hardware and software integration. In this case, these systems are called Real-Time Embedded Systems (RTES) [86]. Considering the possibilities of using the real-time systems cited above, we can see that several RTS are complex and critical, since defects can mean losses in financial, environmental or human areas. For these systems, dependability is an important property that demands rigorous application of Verification and Validation (V & V) activities. As the costs and consequences of failures are high, formal verification and model checking have been used in the V & V process. However, these approaches have practical limitations. In this context, testing arises as an important complementary approach since it allows the execution of real scenarios within execution environments. Furthermore, testing is one of the most popular validation techniques and if used in an effective way may provide important evidences of product quality and reliability. Thus, one of the challenges today is the search for methods, techniques and tools to support the testing of RTES.

Particularly, testing RTES poses a number of distinguishing challenges such as the development platform is usually different from the execution platform, and also there are several execution platforms leading to several cross-development environments. Moreover, RTES are usually composed of parallel activities with synchronous and asynchronous events (interruptions), with different deployment architectures, and resource limitation and time constraints on the execution environment. In this context, interruptions are usually applied so that services can be activated as soon as demanded. For this, the task running in the foreground is instantly suspended to release resources for the interrupting task. After interruption, the interrupted task should resume from the point where it stopped [2; 55; 79; 86]. As an example from the mobile phone context, when a user is composing an e-mail by using a mobile phone device and an incoming call arrives in this device, the call feature interrupts the e-mail feature that must successfully resume later.

Most work on V & V, in the RTS context, are related to the model checking field [13; 31; 54]. Model checking is a technique used to verify, in an automated and accurate manner, the correctness of models. Through the use of models and properties specified in various formalisms, a model checker verifies whether a model satisfies a particular property. However, if the same rigour is not applied to the test of the implementation, a gap is created between these processes, allowing the presence of defects in the implementation even if the model has been successfully verified.

In this sense, several approaches were developed in order to adapt model checking techniques to support test case generation [19; 46; 60; 80]. Furthermore, some classic testing approaches have been extended to support the test of RTS, for instance, model-based testing [73], specially conformance testing [78] and the use of test purposes to select specific scenarios to be verified [17; 87]. Also, some approaches extend finite state machines (FSMs) and their associated methods to deal with time [46; 97].

There are still very few works in this context and most of them use either (variations of) state machines or (variations of) timed automata as the underlying model. However, most approaches to testing real-time systems abstract only time and enumerate data values. This is not suitable when the specification uses large or infinite data domains because data values are enumerated, leading to the state space explosion problem.

In practice, RTS handle variables and action parameters. Thus, powerful models are needed where variables, action parameters and time are explicitly modelled and treated in a symbolic way. There are few works whose goal is to provide symbolic approaches to software testing [27; 32; 52; 63; 64; 65; 72; 85; 108; 113; 114; 120]. However, most of these approaches do not take time requirements into account [27; 32; 52; 63; 64; 65; 72; 85; 108] and the most recent approaches presented in [113; 114; 120] are far from a complete testing approach. The work described in [120] proposes a new symbolic model along with a symbolic conformance relation but neither test cases are formally defined nor algorithms for test case generation are presented. The conformance testing approach proposed in [113; 114] is restricted to data-flow real-time systems making it difficult to test interruptions. Moreover, test cases are not formally defined, algorithms are not presented and there is no tool to support the work.

The rest of this chapter is structured as follows. An overview of this thesis along with its main contributions are presented in Section 1.1. The adopted methodology is described in Section 1.2. Finally, Section 1.3 presents the structure of this thesis.

1.1 Overview of the Thesis

This thesis focuses on symbolic model-based conformance testing of real-time systems where the implementation is a black-box whose internal details are unknown. Thus, the tester can only interact with the implementation through its observable behaviour (inputs and outputs). In this kind of testing, test cases are derived from a formal specification, based on a conformance relation between the implementation and the specification, and used to guide the verdicts of test execution [118]. The verdicts are decided by a component called oracle. In this context, this thesis addresses the following research questions:

Research Question 1 *In which ways can we extend the symbolic model-based testing theory to be able to test real-time systems in an accurate manner?*

Research Question 2 *In a real-time symbolic model-based testing context, how can we provide models to be able to specify and test asynchronous events such as interruptions in an accurate manner?*

Research Question 3 *In a real-time symbolic model-based testing context, is it possible to provide an automated oracle?*

In order to answer these research questions, this thesis proposes a new approach to testing real-time systems. In this sense, our main contributions are:

- A new model-based conformance testing approach to real-time systems is proposed, where the system under test (SUT) is modelled through an extension of both symbolic transition systems and timed automata, thus dealing with both data and time requirements;
- We propose a test case generation process based on test selection using test purposes, which is based on symbolic execution and constraint solving for the data aspects combined with symbolic analysis of timed aspects. Although the proposed symbolic model can represent internal actions and non-determinism, the algorithms defined for test case generation do not take these characteristics into account. Moreover, quiescence and non-input-completeness of implementations are outside the scope of this thesis;
- A strategy to interruption testing is proposed along with a simple way of defining test purposes in order to check specific scenarios with interruptions;
- Our testing approach provides, besides algorithms for test case generation, a complete definition of a test architecture including automatic ways of test execution and reliable verdicts achievements.

The practical application of the proposed approach is evaluated using two case studies. In one case study all test process is performed (from test case generation to test case execution) since an implementation of the system is available. On the other hand, as the second case study does not have an implementation of the system, only the test case generation is considered. The obtained results show that the proposed approach reduces the effort spent to perform the test process since test case generation and evaluation of test execution results are completely automated. However, some points of improvements in the test case execution activity were detected.

It is important to remark that this thesis considers the following assumptions in order to address the defined research questions:

1. The non-real-time symbolic testing theory presented in [32; 63; 64; 65; 108] is solid enough to be extended to deal with timing properties;

2. Symbolic model-based testing provides a good basis for conformance testing of real-time systems.

1.2 Methodology

In general, the methodology used to develop this work is described as follows:

- The first step was to perform a review of work on model-based testing of real-time systems for identifying open problems. It is important to mention that this review was constantly updated during the development of this work;

- The problem of testing interruptions was investigated in a non-real-time context. In this case, an interruption testing approach was proposed to reactive systems;

- Execution of a case study for evaluating the proposed interruption testing approach;

- Symbolic model-based approaches were studied and one approach was chosen as the underlying theory;

- An interruption testing strategy was defined for the chosen non-real-time symbolic model-based approach;

- The chosen symbolic model-based formalism was extended to deal with time requirements and its semantics was formally defined;

- Formalisation of the concept of test case and test purpose;

- An existing conformance relation was chosen to be used in the proposed approach;

- Definition of test case generation and selection algorithms;

- Formalisation of the notion of verdicts considering that a test case execution can produce one of the following verdicts: pass, fail, or inconclusive;

- Definition of strategies for specification and testing of interruptions using the proposed approach;
- Definition of a test architecture and controllability assumptions for test case execution;
- Development of an environment to allow automated test execution and evaluation of results;
- Execution of case studies to evaluate the applicability of the proposed approach.

1.3 Outline of the Thesis

The remaining parts of this document are structured as follows:

Chapter 2: Background This chapter provides the theoretical background necessary to understand this work, including concepts of the software testing field such as test cases, oracles, and testing techniques. Finally, concepts related to real-time systems are presented.

Chapter 3: Interruption Testing of Reactive Systems This chapter presents an approach to conformance testing of reactive systems with interruptions. Real-time systems are not considered yet.

Chapter 4: Related Work and Problem Statements This chapter presents a review of relevant work on testing of real-time systems. Finally, some open problems are described.

Chapter 5: Timed Input-Output Symbolic Transition Systems This chapter proposes the symbolic formalism defined in order to abstract data and time.

Chapter 6: Conformance Testing with TIOSTS This chapter presents the conformance testing theory to deal with the model proposed. Moreover, the test case generation process is described along with a discussion of some properties of the generated test cases.

Chapter 7: Interruption Testing of Real-Time Systems The strategy of modelling and testing interruptions using the proposed symbolic formalism is described in this chapter.

Chapter 8: Case Studies This chapter presents a practical demonstration of the proposed symbolic model-based testing approach to real-time systems.

Chapter 9: Concluding Remarks This final chapter presents the conclusions and prospects for future work.

Chapter 2

Background

This chapter has as main objective to provide a theoretical foundation related to the concepts discussed in this thesis. It discusses concepts from the software testing field, with emphasis on model-based testing and symbolic testing, and concepts from the real-time systems context.

2.1 Software Testing

Software testing is an activity that involves the effort to find evidences of defects inserted into the software during any phase of development or maintenance of software systems. These defects may be due to omissions, inconsistencies or misunderstanding of requirements or specifications by developers [95]. In the software testing context, some concepts are widely used: **failure**, **fault**, and **error**. According to Binder [18], a **failure** is the manifested inability of a system to correctly perform a required function; a **fault** is defined as the absence of code or the presence of incorrect code in a computer program that causes the failure; and **error** is a human action that results in a software fault.

Testing is an important activity that contributes to ensuring that a software system does everything it is supposed to do. Some testing efforts extend the focus to ensure an application does nothing more than it is supposed to do. In any case, testing provides means to assess the existence of defects (faults) which could result in a loss of time, property, customers, or life [95].

For a long time, the software testing process was defined within software development processes as a disconnected activity that was only taken into account at the end of the development processes. This traditional view is considered as being inefficient because of high costs associated with correction of detected errors and maintenance of software. This has contributed to the development of methods and systematic testing techniques where the testing activities are applied in parallel during the development process [95].

There are several kinds of tests that can be applied depending on the property of the systems to be tested (for instance, interface, performance, safety, etc.), and their type (for instance, object-oriented software, distributed systems, reactive systems, real-time systems).

The remainder of this section presents several concepts related to software testing and that are important to understand this thesis, such as test cases, oracles, approaches to identify and generate test cases, conformance testing, and some testing techniques such as model-based testing, property oriented testing, and symbolic testing.

2.1.1 Test Cases

A test case is a set of inputs, execution conditions, and expected results chosen in order to test a particular behaviour of a system [18]. The main key of software testing is to determine a set of test cases (named test suite) for the software system to be tested. Every test case must have at least the following information [67]:

- Inputs
  - Conditions that must be satisfied before the test execution;
  - The actual inputs chosen to test the system;
- Outputs
  - Postconditions that must be satisfied after the test execution;
  - The actual output produced by the system under test.

Moreover, a good test case must present additional information for supporting the testing management [67]. For example, a test case may have a unique identifier, a purpose, an execution history, etc. Considering all this information, the act of testing consists in satisfying the preconditions, providing the test case inputs, observing the outputs, and then comparing these outputs with the expected outputs to decide whether the test pass or not.

In the context of this work, test cases can be classified into two types: **instantiated** and **abstract** test cases. We say that a test case is **instantiated** when the values of all variables needed for the test execution are properly assigned during the test case generation process, whereas we say that a test case is **abstract** when it has variables with unassigned values. In the latter case, the tester must assign values, according to the preconditions, during the test execution.

2.1.2 Oracles

The execution of a test case emits a pass verdict when the system produces an acceptable result. In order to decide which verdict must be emitted, an evaluation is made by comparing the actual result with an expected result. The component responsible for performing this evaluation is called test oracle or simply oracle [18]. Thus, an oracle is a mechanism that applies a pass or fail verdict to a system execution [105]. For this, it is necessary a result generator and a comparator. The former is responsible for generating the expected results for an input and the latter has the objective of checking the actual results against the expected results.

Considering that a test oracle is a generation and comparison mechanism, it can be classified into three types: manual, automated, and partially automated [18]. Considering the manual oracle, both generation and comparison are manually performed. In the automated oracle, both generation and comparison are automatically performed. Finally, in the partially automated oracle, one of the activities is manually performed, whereas the other is automatic.

Several artefacts developed during the development process can support the testing process as an oracle. The system specification can be used as an oracle, a table of examples of inputs and expected outputs or simply the knowledge of how the software system should operate provided by the development team can be also used as an oracle [18].

In practice, manual and partially automated oracles are error-prone. If a system under test fails to provide some functionality in a very common situation (for example, a menu option is in a wrong place, the system is aborted with an exception or it is restarted), then maybe it can be seen to have a fault. But considering an expected output, specified only by an imprecise description in natural language, a tester may fail to notice a failure. To do better, an oracle must be automated. But, due to the semantic gap between specification and real application values, a problem named the oracle problem arises when an automated mechanism to emit verdicts cannot be defined [91; 92].

The concepts of test case, test suite, and oracle can be related based on a formal framework [56; 91]. Let IUT be an implementation of a software system under test whose input domain is D and output domain is X. Let TC be a test case which is defined as a total function from elements of $D' \subseteq D$ to elements of $X' \subseteq X$. Then $\mathit{dom}(TC) = D'$ denotes the domain of TC. Also, let $TC(p)$ be the corresponding expected output for a given input $p \in \mathit{dom}(TC)$ and $\textsc{iut}(p)$ denote the actual result of executing IUT with input p. The TC passes the system IUT if and only if it passes IUT on all inputs in $\mathit{dom}(TC)$, that is, $\textsc{iut}(p) = TC(p)$ for all $p \in \mathit{dom}(TC)$. As $\mathit{dom}(TC)$ is likely to be infinite, a finite test suite $T \subseteq \mathit{dom}(TC)$ needs to be selected. A function O is called an oracle for IUT on TC if for all $p \in D$ [91]:

$$O(p) = \begin{cases} true, & \text{if } \textsc{iut}(p) = TC(p) \\ false, & \text{if } \textsc{iut}(p) \neq TC(p) \\ true, & \text{if } p \notin \mathit{dom}(TC) \end{cases}$$

Considering the oracle problem, it arises because of the limitation in the definition of an effective procedure to compare $\textsc{iut}(p)$ with $TC(p)$, mainly because these values are defined at different levels of abstraction [91].

2.1.3 Test Cases Identification

The two fundamental approaches used to identify test cases are known as structural and functional testing [67]. Each of these approaches has its advantages and disadvantages.

Structural testing, also known as white box testing, is a kind of testing where test cases are identified based on the system implementation. The objective of structural testing is to test procedural details [101]. Because structural testing is based on the implementation, it can test parts of the system that are not in the specification, but, on the other hand, the structural testing fails to identify behaviours which are in the specification, but have not been implemented.

Functional testing (also known as black box testing) is a kind of testing based on the view that the software system can be considered as a function that maps values from its input domain to values in its output domain [67], that is, a kind of testing performed to verify whether, for a given input, the system produces the correct output. Functional testing is performed only based on the specification of the system.

Because functional test cases are only identified based on the specification, they are independent of how the system is implemented, unlike structural test cases, and therefore, even if the implementation of the system is changed, the test cases are still useful. Another important advantage is that testing activities can be performed in parallel with the implementation, contributing to a better understanding and correction of models and specifications from initial stages of the development processes, avoiding late detection of problems, thus reducing the impact and costs associated with the changes.

In addition to the classification of tests following the fundamental approaches, structural and functional, we can make a new distinction with respect to several aspects of the behaviour of the system to be tested. When the specification is defined by models, the approach is called model-based testing [43][1]. When the test is carried out to verify whether the system has the planned functionalities and if those functionalities are in accordance with the specification, it is called conformance testing [117]. When the goal is to test specific properties of the system, test case generation can be guided by informal descriptions of the behaviours to be tested. In this case, the approach is called property oriented testing [62].

2.1.4 Model-Based Testing

In the last decade, perhaps due to the popularization of object-oriented programming and use of models in software engineering, there was a great development of a testing technique known as model-based testing (MBT). MBT is a general term used to name a set of techniques based on models of applications being tested in order to perform activities of test [43]. Such activities can be either generation of test cases or evaluation of test results.

The main activities related to model-based testing, shown in Figure 2.1, are described below:

Build the model: the model is built from the requirements of the software system under

[1] As models are considered as specifications in this thesis, these terms are used interchangeably.

test.

Generate test cases: test cases are extracted from the model with the objective of evaluating whether the system is in accordance with its requirements.

Generate test oracle: the test oracle is generated based on the model. The test oracle is responsible for deciding which outputs indicate the correct behaviour of the system, that is, the expected results.

Run tests: the application is exercised with generated test cases, producing new outputs.

Compare actual outputs with expected outputs: the outputs of the system under test, obtained in the previous step, are compared by the test oracle with the expected outputs.

The process of model-based testing begins when the requirements of the software system are defined. From requirements, a model that represents the expected behaviour of the system is built. After defining the model, the next step is the generation of test cases. The specification of test cases includes, among other information, expected inputs and outputs. Using these inputs, the system is executed and its behaviour is observed. The last step is to compare the obtained outputs with the expected outputs to assess whether or not the system is in accordance with requirements.

One of the main advantages of using MBT is that the generated model can serve as a reference point for communication between all the people involved in the development process. Another important advantage is that the most popular models have a rich theoretical basis that facilitates the generation and automation of the testing process [43].

One drawback of using MBT is the need of knowing the notation of the model and the theoretical basis to take the most of the model chosen. To make the team acquire the necessary knowledge implies investment in training and lack of time, in addition to time spent on the construction of the model [43]. Another disadvantage is the high dependence with respect to the model, that is, as the test activities are carried out based on the model of the system, the quality of testing is directly related to the quality of the model.

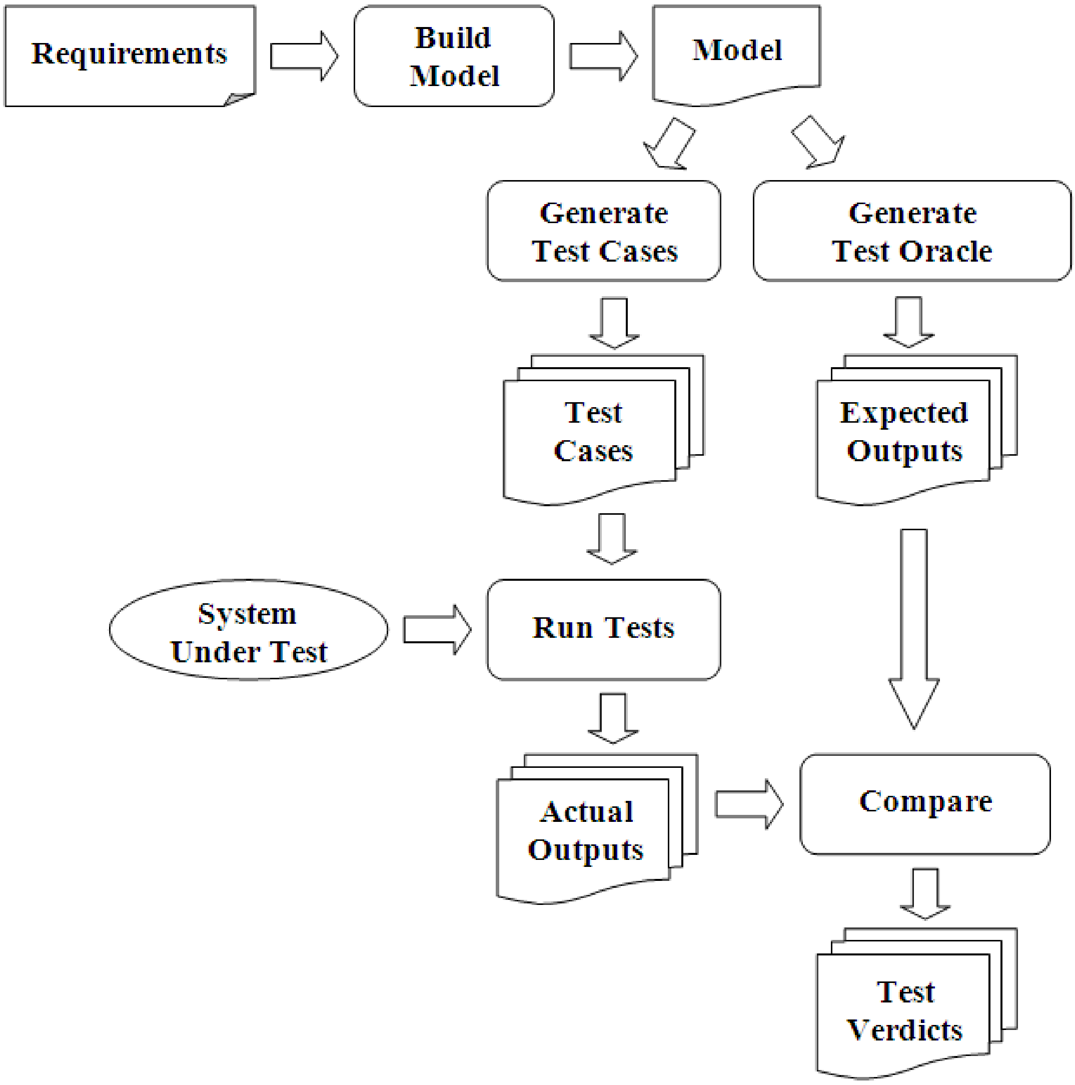


Figure 2.1: MBT Activities

2.1.5 Conformance Testing

Conformance testing is a kind of testing used to verify whether the implementation of a software system is in accordance with the specification of its functional behaviour. So, this subsection presents, in a formal manner, a conformance testing approach based on the framework proposed by Tretmans [118]. Therefore, it is important to link the informal world of implementations and tests with the formal world of specifications and models.

Conformance testing relates a specification and an implementation under test (IUT) by the relation **conforms-to** $\subseteq IMPS \times SPECS$, where $IMPS$ represents the universe of implementations and $SPECS$ represents specifications. Then, IUT **conforms-to** s if and only if IUT is a correct implementation of s.

The **conforms-to** relation is hard to be checked by testing and the implementations are generally unsuitable for formal reasoning. Therefore, a test hypothesis is assumed where any IUT can be modelled by a formal object $i_{\mathrm{IUT}} \in MODS$, where $MODS$ represents the universe of models [15]. Then, an implementation relation **imp** $\subseteq MODS \times SPECS$ is defined such that IUT **conforms-to** s if and only if i_{IUT} **imp** s.

Let $TESTS$ be the domain of test cases and $t \in TESTS$ be a test case. Then $\textsc{exec}(t,\textsc{iut})$ denotes the operational procedure of applying t to IUT. This procedure represents the test execution. Let an observation function that formally models $\textsc{exec}(t,\textsc{iut})$ be defined as $obs : TESTS \times MODS \rightarrow \mathcal{P}(OBS)$. Then, $\forall$ IUT $\in IMPS\ \exists i_{\mathrm{IUT}} \in MODS\ \ \forall t \in TESTS \cdot$ $\textsc{exec}(t,\textsc{iut}) = obs(t, i_{\mathrm{IUT}})$, according to the test hypothesis.

Let a family of verdict functions $v_t : \mathcal{P}(OBS) \rightarrow \{$**fail**, **pass**$\}$ which can be abbreviated to IUT **passes** $t \Leftrightarrow_{def} v_t(\textsc{exec}(t,\textsc{iut})) =$ **pass**. Then, for any test suite $T \subseteq TESTS$, IUT **passes** $T \Leftrightarrow \forall t \in T \cdot$ IUT **passes** t. Also, IUT **fails** $T \Leftrightarrow \neg($IUT **passes** $T)$. A test suite that can distinguish between all conforming and non-conforming implementations is called *complete*. Let $T_s \subseteq TESTS$ be complete. Then, IUT **conforms-to** s if and only if IUT **passes** T_s.

A complete test suite is a very strong requirement for practical testing. Then, weaker requirements are needed. A test suite is *sound* when all correct implementations and possibly some incorrect implementations pass it, that is, any detected faulty implementation is non-conforming, but not the other way around. Let $T \subseteq TESTS$ be sound. Then, IUT **conforms-to** $s \Rightarrow$ IUT **passes** T. The other direction of the implication is called *exhaustiveness*, meaning that all non-conforming implementations will be detected.

In practice, sound test suites are more commonly accepted, since rejection of conforming implementations, by exhaustive test suites, may lead to unnecessary debugging. Let $der_{imp} : SPECS \rightarrow \mathcal{P}(TESTS)$ be a test suite derivation algorithm. Then, $der_{imp}(s)$ should only produce sound and/or complete test suites.

This testing framework is instantiated by several works using the most different notations. For instance, Tretmans [118] instantiated the framework with Labelled Transition Systems (LTS), Jard and Jéron [62] instantiated it using Input-Output Labelled Transition Systems (IOLTS), Larsen et al. [80] and Krichen [73] instantiated the framework using Timed Input-Output Transition Systems (TIOTS), Briones and Brinksma [23] extended the Tretmans' framework for real-time systems, etc.

One of the most important properties considered in conformance testing is called quiescence. In practice, tests observe the behaviour of the system and its quiescence, that is, the absence of outputs. Quiescence is observed using timers, for instance, whenever a tester sends an input to the implementation, a timer is reset. The duration of the timer is chosen such that, if no output occurs while the timer is running, then no output will ever occur. Then, when the timer finishes, the tester can conclude that the implementation is quiescent. This approach avoids the rejection of implementations expected to be quiescent in some points and rejects the implementations that are not, ensuring the soundness of the test cases.

In order to distinguish between observations of quiescence that are allowed by a specification and those that are not, all possible points where an implementation may become quiescent must be made explicit in the specification. There are three possibilities of quiescence: deadlock, output quiescent, and livelock [62]. A deadlock state is a state where the system cannot evolve anymore. An output quiescent state is a state where the system is waiting only for an input from the environment. And, livelock is related to the loops of internal actions, that is, loops of actions that are not seen from the environment.

2.1.6 Property Oriented Testing

It is important to note the difference between testing for conformance and testing from test purposes. The former aims to accept/reject a given implementation. On the other hand, the latter aims to observe a desired behaviour that is not necessarily directly related to a required behaviour or correctness. If this desired behaviour is observed then confidence on correctness may increase. Otherwise, no definite conclusion can be based solely on this information. Due to its overloaded use, test purpose is called observation objective in [41]. Nevertheless, the term test purpose is kept in thesis. The concepts introduced in Subsection 2.1.5 are extended for test purposes in this subsection.

Test purposes describe desired observations that we wish to see from the implementation during the test. Test purposes are related to implementations that are able to exhibit them by a well chosen set of experiments. This is defined by the relation **exhibits** $\subseteq IMPS \times TOBS$, where $TOBS$ is the universe of test purposes. To reason about exhibition, we also need to consider the test hypothesis from Subsection 2.1.5 by defining the *reveal* relation **rev**

$\subseteq MODS \times TOBS$, so that, for $e \in TOBS$, IUT **exhibits** e if and only if i_{IUT} **rev** e, with $i_{\text{IUT}} \in MODS$ of IUT.

Let a verdict function $H_e : \mathcal{P}(OBS) \to \{$**hit**, **miss**$\}$ which can decide whether a test purpose is exhibited by an implementation. Then, IUT **hits** e **by** $t_e =_{def} H_e(\text{EXEC}(t_e,\text{IUT})) =$ **hit**. This is extended to a test suite T_e as IUT **hits** e **by** $T_e =_{def} H_e(\bigcup\{\text{EXEC}(t,\text{IUT}) \mid t \in T_e\}) =$ **hit**, which differs from the **passes** abbreviation.

An *e-complete* test suite can distinguish among all exhibiting and non-exhibiting implementations, such that, IUT **exhibits** e if and only if IUT **hits** e **by** T_e. An *e-exhaustive* test suite can only detect non-exhibiting implementations (IUT **exhibits** e implies IUT **hits** e **by** T_e), whereas an *e-sound* test suite can only detect exhibiting implementations (IUT **exhibits** e if IUT **hits** e **by** T_e). Note that the purpose of the *sound* test suites and *e-sound* test suites are similar, even though the implications are relatively inverted. *Sound* test suites can reveal the presence of faults, whereas the *e-sound* can reveal intended behaviour.

Conformance and exhibition can be related aiming to consider test purposes in test selection to obtain test suites that are sound and e-complete. We want *e-soundness* so that we can conclude that a hit result always implies exhibition, whereas we require *e-exhaustiveness* because we want to be able to find all implementations that are able to exhibit. Soundness provides us with the ability to detect non-conforming implementations. Contrary to complete test suites, *e-complete* test suites are feasible.

Finally, it is important to remark that both conforming and non-conforming implementations may reveal a test purpose. An ideal situation, where all correct implementations also exhibit, would be to only consider a test purpose e when i **rev** $e \supseteq i$ **passes** T, where $T \subseteq TESTS$. However, this situation is not practical. Test purposes are chosen so that: $\{i \mid i$ **rev** $e\} \cap \{i \mid i$ **imp** $s\} \neq \emptyset$. In this case, a test execution with test case $T_{s,e}$ that is both sound and e-complete and that results in **fail** means non-conformity, since sound test cases do not reject conforming implementations and e-complete test cases distinguish between all exhibiting and non-exhibiting implementations. Also, if the result is $\{$**pass**, **hit**$\}$, confidence on correctness is increased, as the hit provides possible evidence of conformance.

2.1.7 Test Case Generation

The test case generation activity can be done following two different approaches: offline or online testing. In offline testing, all test cases are extracted from the specification, after that, they are executed against the implementation to obtain a verdict.

The other approach to test case generation is online (on-the-fly) testing where the generation and execution are combined into a single step. In this approach, a single action or input is extracted from the specification at a time and it is immediately executed on the implementation. Then the output produced by the implementation is checked against the specification. After that, another action or input is extracted again and so forth until the end of the test, or until a defect (fault) is detected.

As advantages, the offline test generation can be guided to generate test cases in order to reach some objective, e.g. specification structural coverage, test specific behaviours with test purposes, and so on. During the offline test generation, all time constraints are resolved before the execution, so the generated test cases are cheaper and faster to execute [59].

One of the main advantages of online testing is that the classical state space explosion problem is reduced because only a limited part of the state space needs to be stored at any point in time, whereas the state space explosion problem is very common when the offline approach is adopted because the state space needs to be entirely built and stored. Another advantage of online testing is that some important characteristics of real implementations as non-determinism can be treated during the test execution more easily. However, online testing may not be applicable in environments with limited resources, for example in the test of some embedded systems (e.g. smart cards, mobile phones, music players, etc), because of the need of very efficient test generation algorithms and many resources to execute them.

2.1.8 Symbolic Execution

Symbolic execution is a technique for analysing programs which represents program inputs with symbolic values instead of concrete values [33; 71]. The execution of programs is simulated by manipulating expressions involving symbolic values. Thus, the outputs are expressed as a function of the symbolic inputs. This technique is used in different contexts such as test input generation, reachability analysis, partial correctness proving of programs, Algorithm 2.1: Code for Swapping Two Integers Variables

```
1 int x, y;
2 if (x > y) {
3   x = x + y;
4   y = x - y;
5   x = x - y;
6   if (x - y > 0) {
7     assert(false);
8   }
9 }
```

etc.

The execution paths of a program identified during its symbolic execution are represented as a symbolic execution tree, whose nodes are the program states connected by program transitions. A program state includes the symbolic values of program variables, a path condition (PC), and a program counter. A path condition is a (quantifier-free) boolean formula over the symbolic inputs. When a PC is satisfiable means that it is possible to reach the specific program point associated with it. Otherwise, the referred specific program point is unreachable.

Consider the code fragment in Algorithm 2.1, which swaps the values of integer variables x and y when x is greater than y, with its corresponding symbolic execution tree presented in Figure 2.2, where transitions are labelled with program statement line numbers [104]. At the beginning, *PC* is true and x and y have X and Y as symbolic values, respectively. At each statement of the code, *PC* is updated according to conditions associated with variables. Analysing the symbolic execution tree of Figure 2.2, it is possible to conclude that line 7 of Algorithm 2.1 is unreachable because the corresponding *PC* is not satisfiable. In practice, constraint solvers are used to verify whether a path condition is satisfiable.

2.1.9 Symbolic Testing

In the last years, several theories and techniques of test case generation have been developed through specifications modelled by variations of the classic LTS [12; 29; 40; 62; 85; 117]. Basically, LTS models and its variations represent a system behaviour through a graph where

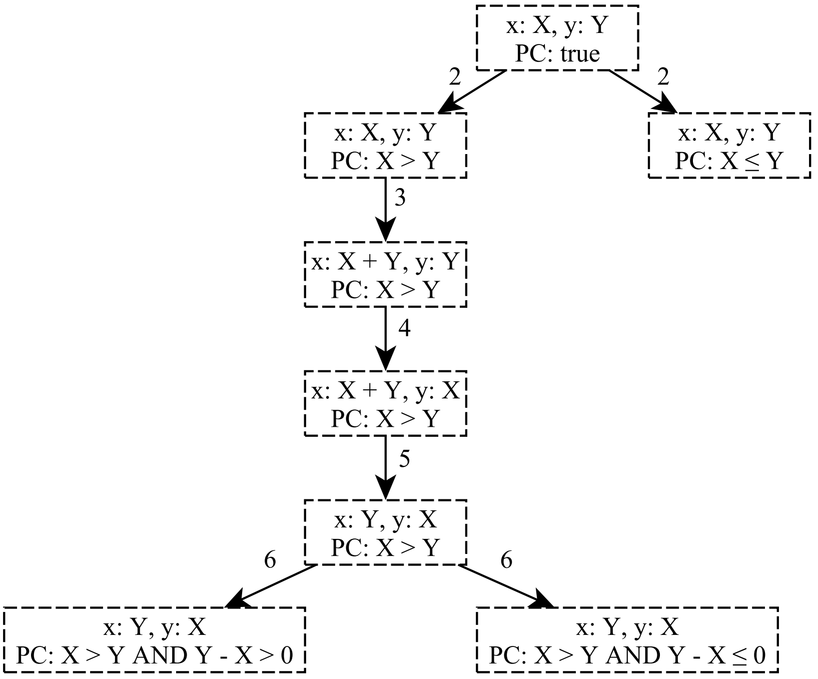


Figure 2.2: Symbolic Execution Tree of Algorithm 2.1

the states are the possible system configurations and the edges represent the action of moving between these configurations through actions occurrence.

However, LTS models are not suitable when the specification uses large or infinite data domains because each value in the data domain is represented as a system state, leading to the classical state space explosion problem. Consequently, many tools can only be used in very restricted and finite domains.

In practice, test cases (written, for example, using TTCN [49]) can be real programs with parameters and variables. In this context, a new approach to testing arises: symbolic testing. Symbolic testing is a testing approach based on powerful models where variables and parameters are explicitly modelled and treated in a symbolic way.

In the context of symbolic models, the principle is to adapt some existing approach to, beyond representing the system behaviour, represent the system data without enumerating the data values. There are still very few works in this context and most of them use (variations of) state machines or labelled transition systems as the underlying model.

State machines tend to be used in synchronous contexts, where inputs and outputs appear together in a single transition. Thus, they are unsuitable for representing some characteristics of reactive systems such as non-determinism and interruptions. Considering the use of state machines, there is a tool named GAST [72] that was extended to deal with Extended Finite State Machines (EFSM) specifications. In this tool, properties and data types are expressed in first-order logic, and based on this information, test data is automatically generated. This tool is less suitable for testing because the concept of state is not present in a clear manner, even though it is possible to represent states defining explicitly a complex data structure that represents the state space. GAST's algorithm unfolds, in an on-the-fly way, the data type structure in order to select a path in the EFSM.

The use of (variations of) labelled transition systems is more common in the literature. Lestiennes and Gaudel [85] developed a strategy of test generation and selection based on selection hypotheses combined with an operation of unfolding algebraic data types and predicate resolution. Frantzen et al. [52] extended the theory presented in [117] to support software testing based on symbolic models. The symbolic framework developed by Frantzen et al. uses concepts from first-order logic as underlying theory for dealing with guards and variables, quiescence is taken into account, and some ideas about coverage is discussed. However, the symbolic framework does not consider test purposes and there is no tool support. Calamé et al. [27] proposes an approach combining symbolic models, data abstraction, and constraint solving to generate test cases. The main idea is to apply data abstraction to abstract the model in a finite state one, use the TGV tool [62] to generate abstract test cases, and finally, constraint solving is applied to instantiate the test cases.

One of the most solid approaches in the context of symbolic testing is presented in [32; 63; 64; 65; 108]. This method works directly on high-level specifications given as Input-Output Symbolic Transition Systems (IOSTS) without enumerating their state space. Test purposes are taken into account to verify specific behaviours of an implementation. Approximate coreachability analysis is used to prune paths potentially not leading to pass verdicts [63]. The coreachability analysis is based on Abstract Interpretation [36] and the concept of test generation with verification techniques is also based on the theory presented in [62; 117]. Finally, constraint solving is applied to instantiate the test cases. Moreover, all the symbolic testing approach is supported by the STG tool [32]. So, the theory related to IOSTS will be presented in more detail below.

The IOSTS is a model of extended labelled transition systems that was inspired by I/O automata [89]. An IOSTS is a symbolic automata with a finite set of locations, typed variables, and the communication with its environment is performed through actions carrying parameters.

Definition 2.1 (IOSTS). *Formally, an IOSTS is a tuple $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$, where [108]:*

- *V is a finite set of typed variables;*
- *P is a finite set of parameters. For $x \in V \cup P$, $type(x)$ denotes the type of x;*
- *Θ is the initial condition, a predicate with variables in $V \cup P$;*
- *L is a finite, non-empty set of locations;*
- *$l^0 \in L$ is the initial location;*
- *$\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is a finite, non-empty alphabet, where $\Sigma^?$ is a finite set of input actions, $\Sigma^!$ is a finite set of output actions, and Σ^τ is a finite set of internal actions. Each action $a \in \Sigma$ has a signature $sig(a) = \langle p_1, ..., p_k \rangle$, that is a tuple of distinct parameters. The signature of internal actions is the empty tuple;*
- *$\mathcal{T}$ is a set of transitions, where each transition consists of:*
  - *a location $l \in L$, called the origin of the transition,*
  - *an action $a \in \Sigma$, called the action of the transition,*
  - *a predicate G with variables in $V \cup P \cup sig(a)$, called the guard,*
  - *an assignment A, such that for each variable $x \in V$ there is exactly one assignment in A, of the form $x := A^x$, where A^x is an expression on $V \cup P \cup sig(a)$,*
  - *a location $l' \in L$, called the destination of the transition.*

$\diamond$

Figure 2.3 shows an example of an IOSTS. In graphical representations, input actions are followed by the “?” symbol and output actions are followed by the “!” symbol. These symbols are used only as notation, they are not part of the action’s name. The simple IOSTS depicted in Figure 2.3 models the triangle problem, an example widely used in the literature [67; 101; 106]. The input of the problem is three integers representing the sides of a triangle and the output is the type of the triangle. At the beginning, the system is in the

Idle location. Next, the system expects the *Read* input carrying three strictly positive integer parameters *p*, *q*, and *r*. Then the values of the parameters are saved into the variables *a*, *b*, and *c*, respectively. If the values of the variables do not represent a triangle the system leaves the *CheckTriangle* location and goes to the *End* location emitting the *NotATriangle* output. Otherwise, the system emits the *IsTriangle* output followed by the type of the triangle (equilateral, isosceles, or scalene).

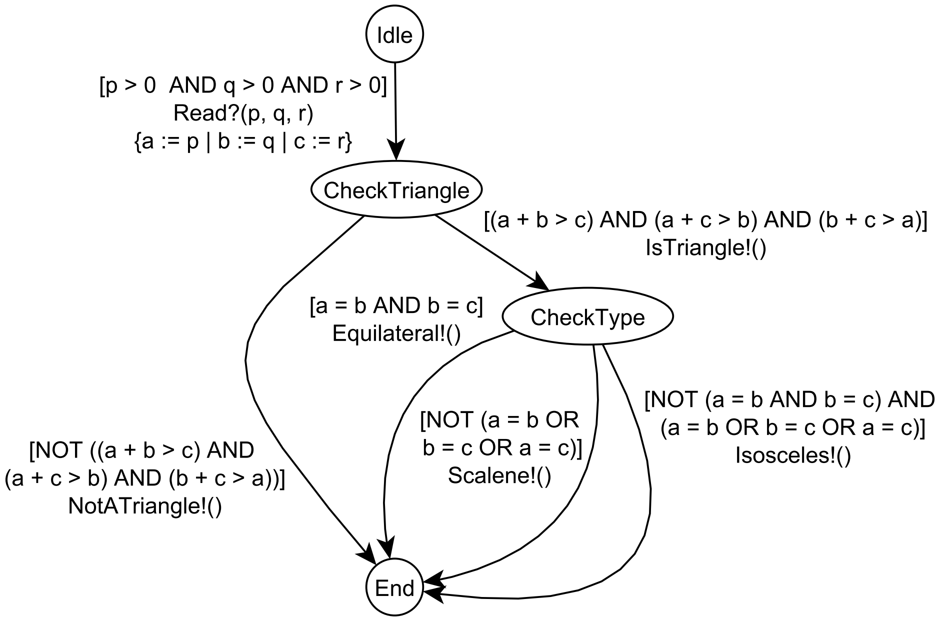

Idle
[p > 0 AND q > 0 AND r > 0]
Read?(p, q, r)
{a := p | b := q | c := r}
CheckTriangle
[(a + b > c) AND (a + c > b) AND (b + c > a)]
IsTriangle!()
[a = b AND b = c]
Equilateral!()
CheckType
[NOT (a = b AND b = c) AND
(a = b OR b = c OR a = c)]
Isosceles!()
[NOT (a = b OR
b = c OR a = c)]
Scalene!()
[NOT ((a + b > c) AND
(a + c > b) AND (b + c > a))]
NotATriangle!()
End

Figure 2.3: IOSTS Example

The semantics of IOSTS is defined through *Input-Output Labelled Transition Systems* (IOLTS) [35]. An IOLTS is a variant of the classic LTS that makes distinction between events of the system that are controllable by the environment (the inputs) and those that are only observable (the outputs) [117]. Moreover, internal actions can be represented too.

Definition 2.2 (IOLTS). *An IOLTS is a tuple $\langle Q, Q^0, \Lambda, \rightarrow \rangle$, where [35]:*

- *Q is a possibly infinite set of states;*
- *$Q^0 \subseteq Q$ is the possibly infinite set of initial states;*
- *$R = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ is a possibly infinite set of actions, where $\Lambda^?$ is the set of input actions, $\Lambda^!$ is the set of output actions, and Λ^τ is the set of internal actions;*

- *$\rightarrow \subseteq Q \times \Lambda \times Q$ is the transition relation.*

$\diamond$

Intuitively, the IOLTS semantics of an IOSTS $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ enumerates the possible values of the variables V and parameters P through the *valuations* of their domains. A *valuation* of the variables V is a mapping ν which maps every variable $x \in \mathcal{V}$ to a value $\nu(x)$ in the domain of x. Valuations of parameters P are defined similarly.

Let $\mathcal{V}$ denote the set of valuations of the variables V and let Γ denote the set of valuations of the parameters P. Considering $\nu \in \mathcal{V}$ and $\gamma \in \Gamma$, an expression E involving a subset of $V \cup P$, denoted by $E(\nu, \gamma)$, is the value obtained by evaluating the result of the replacement in E of each variable by ν and each parameter by γ.

Definition 2.3 (IOLTS semantics of an IOSTS)**.** *The semantics of an IOSTS $S = \langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ is an IOLTS $\mathcal{S} = \langle Q, Q^0, \Lambda, \rightarrow \rangle$, defined as follows [35]:*

- *$Q = L \times \mathcal{V}$ is the set of states;*
- *$Q^0 = \{\langle l^0, \nu \rangle \mid \Theta(\nu) = true\}$ is the set of initial states;*
- *$\Lambda = \{\langle a, \gamma \rangle \mid a \in \Sigma, \gamma \in \Gamma_{sig(a)}\}$ is the set of actions, where Λ is partitioned into the sets $\Lambda^?$ of input actions, $\Lambda^!$ of output actions, and Λ^τ of internal actions;*
- *$\rightarrow$ is the smallest relation in $Q \times \Lambda \times Q$ defined by the following rule:*

$$\frac{\langle l, \nu \rangle, \langle l', \nu' \rangle \in Q \quad \langle a, \gamma \rangle \in \Lambda \quad t : \langle l, a, G, A, l' \rangle \in \mathcal{T} \quad G(\nu, \gamma) = true \quad \nu' = A(\nu, \gamma)}{\langle l, \nu \rangle \overset{\langle a, \gamma \rangle}{\rightarrow} \langle l', \nu' \rangle}.$$

$\diamond$

Intuitively, the rule says that the system moves from a state $\langle l, \nu \rangle$ to a state $\langle l', \nu' \rangle$ through an action $\langle a, \gamma \rangle$ if there is a transition $t : \langle l, a, G, A, l' \rangle$ whose guard G is evaluated to $true$. Finally, the assignment A maps the pair (ν, γ) to ν'.

Next, we present some important definitions used to define the conformance relation between IOSTS specifications and the implementation of the system under test.

Definition 2.4 (Traces)**.** *Let L^0 denote the set of initial states. For an IOSTS $\mathcal{R}$ we denote by $traces(\mathcal{R})$ the set $\{\sigma \in (\Sigma^? \cup \Sigma^!)^* \mid \exists l^0 \in L^0, \exists l \in L, l^0 \overset{\sigma}{\Rightarrow} l\}$.* $\diamond$

Definition 2.5 (After). *For $\sigma \in (\Sigma^? \cup \Sigma^!)^*$, we denote by $\mathcal{R}$ after σ the following set of states: $\{l \in L \mid \exists l^0 \in L^0, l^0 \overset{\sigma}{\Rightarrow} l\}$.* ◇

Definition 2.6 (Out). *For $L' \subseteq L$ be a set of states, we denote by $out(L')$ the set of valued outputs that can be observed in states $l' \in L'$, that is, $out(L') = \{\alpha \in \Omega \mid \exists l' \in L',$ $\exists l \in L,\ l' \overset{\alpha}{\Rightarrow} l\}$.* ◇

Definition 2.7 (Pref). *For a set of traces T, we denote by pref(T) the set of strict prefixes of sequences in T.* ◇

Next, the formal framework for conformance testing presented in [118] (Subsection 2.1.5) and the formal framework for test purposes presented in [41] (Subsection 2.1.6) are instantiated. For this, the following concepts related to the frameworks must be defined: specifications, implementations, test purposes, test cases, verdicts, and the conformance relation.

Specifications. A specification is an IOSTS without cycles of internal actions.

Implementations. An implementation can be any computer system that can be modelled by an IOSTS.

Test Purposes. A test purpose is an IOSTS that describes a specific scenario to be verified. Before seeing the formal definition of a test purpose it is important to know two characteristics of IOSTS: completeness e compatibility.

Definition 2.8 (Completeness). *An IOSTS is complete if for each $l \in L, \alpha \in \Sigma$, the set $\{l' | l \xrightarrow{\alpha} l'\}$ is non-empty, that is, each location allows all actions.* ◇

Definition 2.9 (Compatibility). *Let $S_1 = \langle V_1, P_1, \Theta_1, L_1, l_1^0, \Sigma_1, \mathcal{T}_1 \rangle$ and $S_2 = \langle V_2, P_2, \Theta_2, L_2, l_2^0, \Sigma_2, \mathcal{T}_2 \rangle$ be two IOSTS. We say that S_1 and S_2 are compatible if $V_1 \cap V_2 = \emptyset$, $P_1 = P_2$, $\Sigma_1^! = \Sigma_2^!$, $\Sigma_1^? = \Sigma_2^?$, and $\Sigma_1^\tau \cap \Sigma_2^\tau = \emptyset$.* ◇

Definition 2.10 (Test Purpose). *Let $S = \langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ be an IOSTS. A test purpose of S is an IOSTS $\mathcal{TP} = \langle V_{\mathcal{TP}}, P_{\mathcal{TP}}, \Theta_{\mathcal{TP}}, L_{\mathcal{TP}}, l_{\mathcal{TP}}^0, \Sigma_{\mathcal{TP}}, \mathcal{T}_{\mathcal{TP}} \rangle$ together with a set of locations $Accept_{\mathcal{TP}} \subseteq L_{\mathcal{TP}}$ and $Reject_{\mathcal{TP}} \subseteq L_{\mathcal{TP}}$ such that $\mathcal{TP}$ is complete and compatible with S.* ◇

Figure 2.4 presents an example of a test purpose for the triangle problem example. It is used to select scenarios where the user chooses inputs such that the triangle is equilateral.

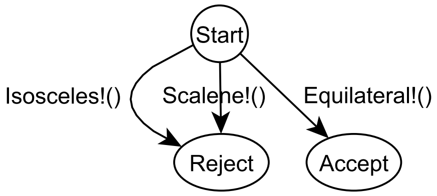


Figure 2.4: Test Purpose Example

The *Reject* location is used to discard all other scenarios where the system does not exhibit the desired behaviour. This test purpose is not complete but it is implicitly completed by the test case generation tool, so the activity of defining test purpose is simplified by allowing to focus only on the desired behaviour.

Test Cases. Test cases are used to assign verdicts to implementations.

Definition 2.11 (Test Case)**.** *A test case is an input-complete, deterministic IOSTS with three disjoints sets of locations:* Pass*,* Inconclusive*, and* Fail*.* ◇

An example of a test case is showed in Figure 2.5. It starts by providing three integer values to an implementation of the triangle problem example. Then it expects to receive a message informing that the chosen values represent the sides of a triangle. Next, if the system says that the triangle is equilateral, the verdict is **Pass**, that is, the implementation is in conformance with the specification and the test purpose. If some other response allowed

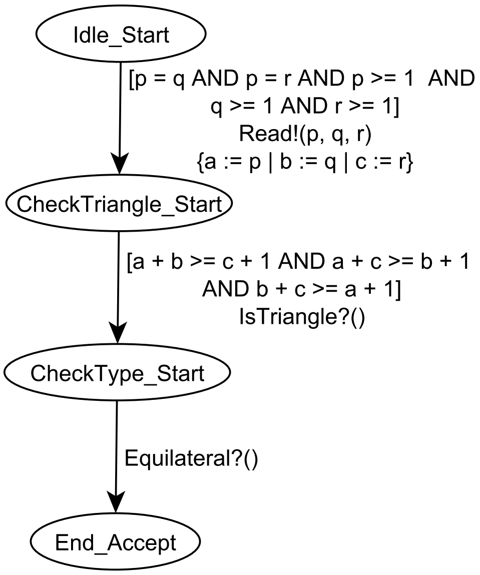


Figure 2.5: Test Case Example

by the specification is emitted then the verdict is **Inconclusive**. Finally, if an unspecified output is emitted then the verdict is **Fail**.

Conformance. The conformance relation links an implementation to the specification and the test purpose. In order to formally define the conformance relation it is needed to define a product operation that identifies in the specification all possible traces obtained with the specified test purpose.

Definition 2.12 (Product)**.** *The product $\mathcal{P} = \mathcal{S}_1 \times \mathcal{S}_2$ of two compatible IOSTS $\mathcal{S}_1$, $\mathcal{S}_2$ is the IOSTS $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ defined by: $V = V_1 \cup V_2$, $P = P_1 = P_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $L = L_1 \times L_2$, $l^0 = \langle l_1^0, l_2^0\rangle$, $\Sigma^? = \Sigma_1^? = \Sigma_2^?$, $\Sigma^! = \Sigma_1^! = \Sigma_2^!$, $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$. $\mathcal{T}$ is the smallest set of transitions satisfying the following rules [35]:*

1. $\dfrac{\langle l_1,a,G_1,A_1,l_1'\rangle \in \mathcal{T}_1,\;\; a \in \Sigma_1^\tau,\;\; l_2 \in L_2}{\langle\langle l_1,l_2\rangle,a,G_1,A_1 \cup (x{:=}x)_{x \in V_2},\langle l_1',l_2\rangle\rangle \in \mathcal{T}}$ *and* $\dfrac{\langle l_2,a,G_2,A_2,l_2'\rangle \in \mathcal{T}_2,\;\; a \in \Sigma_2^\tau,\;\; l_1 \in L_1}{\langle\langle l_1,l_2\rangle,a,G_2,A_2 \cup (x{:=}x)_{x \in V_1},\langle l_1,l_2'\rangle\rangle \in \mathcal{T}}$

2. $\dfrac{\langle l_1,a,G_1,A_1,l_1'\rangle \in \mathcal{T}_1 \quad \langle l_2,a,G_2,A_2,l_2'\rangle \in \mathcal{T}_2}{\langle\langle l_1,l_2\rangle,a,G_1 \wedge G_2,A_1 \cup A_2,\langle l_1',l_2'\rangle\rangle \in \mathcal{T} \quad (for\ a \in \Sigma^! \cup \Sigma^?)}$

◇

The Rule 1, defined above, allows internal actions to evolve independently in each IOSTS and Rule 2 allows the synchronization of the observable actions of the two IOSTS.

Definition 2.13 (Atraces)**.** *Let $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$ be the product of the specification $\mathcal{S}$ with a test purpose $\mathcal{TP}$. Then $Atraces(\mathcal{P})$ is the set of traces of the specification that are selected according to the test purpose.* ◇

Definition 2.14 (Conformance Relation)**.** *Let $\mathcal{S}$ be a specification modelled by an IOSTS, $\mathcal{TP}$ be a test purpose for $\mathcal{S}$, and $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$ their product.*

1. *An implementation I is in conformance with the specification $\mathcal{S}$, denoted by $I\ conf\ \mathcal{S}$, if for all traces $\sigma \in traces(\mathcal{S}) : out(I\ after\ \sigma) \subseteq out(\mathcal{S}\ after\ \sigma)$.*

2. *An implementation I is in conformance with the specification $\mathcal{S}$ and the test purpose $\mathcal{TP}$, denoted by $I\ conf_{\mathcal{TP}}\ \mathcal{S}$, if for all traces $\sigma \in pref(Atraces(\mathcal{S} \times \mathcal{TP}))$: $out(I\ after\ \sigma) \subseteq out(\mathcal{S}\ after\ \sigma)$.*

◇

Intuitively, an implementation conforms to a specification if for all traces of the specification, the set of output actions of the implementation is contained in the set of output actions of the specification. In the second case, an implementation conforms to a specification and a test purpose if the same inclusion holds for each prefix of a trace of the product between specification and test purpose.

2.2 Real-Time Systems

At the same proportion in which computer systems become increasingly complex and ubiquitous in our lives, applications with time restrictions have become increasingly common. Among some examples of these types of applications we can cite military systems, control systems for chemical and nuclear industries, multimedia systems, air traffic control systems, elevator control systems, monitoring of patients in hospitals, embedded systems in robots, cars, airplanes, etc. All these applications have an important characteristic in common: time requirements. For such systems with explicit time requirements, the correct behaviour depends not only on the correctness of the results but also depends on the time at which they are produced [31]. Computer systems with this kind of restrictions are known as Real-Time Systems (RTS).

The classical view of computer systems is that, at some level of abstraction, they are regarded as a black box that receives inputs and provides appropriate outputs, finishing its execution after that (e.g., compilers, numerical analysis applications, and so on). However, most of current computer systems constantly interact with the environment around them continuously sending responses to input stimuli from the environment. These systems are characterized by their executions never finishing and are known as Reactive Systems. In general, RTS fit in the concept of Reactive Systems. In this context, we can say that RTS are computer systems that respond to input stimuli, originated from the environment in which they are inserted, in specific times [79].

According to the level of compliance with time requirements, real-time systems can be classified into Soft Real-Time Systems and Hard Real-Time Systems. Delays related to compliance with a response-time constraint (deadline) can be tolerated in a soft real-time system. On the other hand, systems in which failure to meet response-time constraints may lead to complete and catastrophic system failure are called hard real-time systems [79].

Soft real-time systems are typically used in a scenario where some elements concurrently interact to produce outputs due to input stimuli, for instance, some packets can be dropped in audio or video applications. In this case, violation of constraints results in degraded quality, but the system can continue to operate. A word-processing application is another example since it should respond to stimuli within a reasonable amount of time or its use will be impractical.

Hard real-time systems are used when it is essential to react to an event within a strict deadline. This kind of system requires strong time requirements because the loss of a deadline can mean losses in financial, environmental, or human terms. As examples of hard real-time systems we can cite medical systems such as heart pacemakers, chemical and nuclear industrial process controllers, a car engine control system, and so on. Hard real-time systems are typically found interacting at a low level with physical hardware, in embedded systems.

2.2.1 Modelling Time

In the study of real-time systems, one essential question is the nature of time. Specifying timing properties is difficult and may take different focuses. The focus considered in this work is that time can be classified into discrete or dense time [5].

The discrete or digital-time model considers time as being a monotonically increasing sequence of integers. This model allows to quantitatively express the distance between two events and establish total orders between them with a high level granularity. One of the advantages of this model is that its transformation to other formal languages is easier. One of the disadvantages is that the events of the real world do not always happen at integer-valued times.

In a dense or analogue-time model time increase monotonically as a sequence of real numbers. This model also allows quantitatively express the distance between two events but in a low level granularity. This model is more natural to represent events of the real world because everything happens in a continuous time. One disadvantage is that dense-time models are not simple to transform to a formal language and they are harder to analyse than in the discrete case.

2.2.2 Events

The nature of events is another important concept related to the study of real-time systems. Considering software systems in general, a change in state results in a change in the flow of control of the system. This change in the flow of control can be triggered by commands like if-then-else, case, invocation of procedures or methods, and so on. Thus, an event is any occurrence that causes a change in the flow of control of a software system [79].

Considering the context of real-time systems, events can be classified into synchronous and asynchronous events. The former are those that occur at predictable points in the flow of control and are represented by conditional branches, invocation of procedures or methods, occurrence of internal trap interruptions (in the case of exception handling), etc. The latter occur at unpredictable points in the flow of control. An important characteristic of the asynchronous events is that they are usually caused by external sources, for instance, an alarm system of a building has sensors to detect intruders and once a movement has been detected, the sensors interrupt the main application of the alarm system. In this scenario, the main application of the alarm system cannot predict when an event will occur because it is caused by external sources.

Synchronous and asynchronous events can be classified into periodic and aperiodic events [79]. Considering the alarm system example cited above, as the events do not occur at regular intervals they are called aperiodic asynchronous events. When interruptions are generated by a periodic external clock they can be classified as periodic asynchronous events. A periodic synchronous event is one represented by a sequence of invocation of tasks in a cyclic code, for instance a cyclic invocation of a method. A conditional branch that is not part of a code block (e.g. garbage collection) represents an aperiodic synchronous event.

2.2.3 Modelling Real-Time Systems

There are several formalisms in the literature for modelling real-time systems: timed automata [5], timed CSP [107], time Petri nets [16], real-time logics [22], and timed extended finite state machines [96]. This subsection introduces the most used formalisms: timed labelled transition systems and timed automata with their variations.

Timed Labelled Transition Systems (TLTS) is the simplest model (Definition 2.15). It

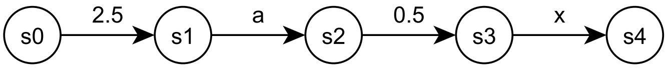


Figure 2.6: TLTS Example

is an extension of the classic LTS where the actions are divided into discrete and time-elapsing actions. The difference between discrete and time-elapsing actions is that the former occurs instantaneously, i.e. without consuming time, whereas the latter represents the time evolution.

Definition 2.15 (TLTS). *Formally, a TLTS is tuple $\langle S, s_0, Act, T\rangle$, where:*

- *S is a finite, non-empty set of states;*
- *$s_0 \in S$ is the initial state;*
- *$Act = A \cup D$ is a set of actions, where A is a finite set of discrete actions and $D = \{d \mid d \in \mathbb{R}^{\geq 0}\}$ is a set of time-elapsing actions;*
- *$T \subseteq S \times Act \times S$ is the transition relation with the following properties:*
  - ***Time Determinism:** $\forall s, s', s'' \in S$: if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$, then $s' = s''$*
  - ***Time Additivity:** $\forall s, s'' \in S$, $\forall d_1, d_2 \geq 0 : (\exists s' \in S : s \xrightarrow{d_1} s' \xrightarrow{d_2} s'')$ iff $s \xrightarrow{d_1+d_2} s''$*
  - ***Null Delay:** $\forall s, s' \in S : s \xrightarrow{0} s'$ iff $s = s'$.*

◇

An example of a TLTS is shown in Figure 2.6. That example represents a scenario where, starting in state s_0, we delay for 2.5 units of time, after which we reach s_1, where we immediately execute action a, reach s_2, delay for 0.5 units of time, after which we reach s_3, where we immediately execute x, and reach state s_4.

The Timed Input-Output Labelled Transition System (TIOLTS) is an extension of TLTS where the set of discrete actions A is partitioned into input and output actions.

Definition 2.16 (TIOLTS). *A TIOLTS is a timed labelled transition system $\langle S, s_0, Act, T\rangle$ with $Act = A_I \cup A_O \cup D$ ($A_I \cap A_O = \emptyset$), where A_I is a finite set of input actions and A_O is a finite set of output actions.* ◇

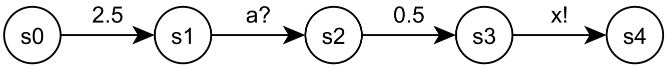


Figure 2.7: TIOLTS Example

Figure 2.7 presents an example of TIOLTS. It means that, starting in state s_0, we delay for 2.5 units of time, after which we reach s_1, where we immediately provide the input action a to the system, reach s_2, delay for 0.5 units of time, after which we reach s_3, where the system immediately responds with the output action x, and reach state s_4.

Most of the work related to real-time model checking and testing is based on Timed Automata (TA). It was firstly proposed by Alur and Dill [5] and ever since many variations have been proposed.

Definition 2.17 (TA). *A Timed Automaton is a tuple $\langle Q, q_0, Act, C, E\rangle$, where:*

- *Q is a finite set of locations;*
- *$q_0 \in Q$ is the initial location;*
- *Act is a finite set of actions;*
- *C is a finite set of clocks;*
- *E is a finite set of transitions. Each transition is a tuple $\langle q, q', a, \lambda, \delta\rangle$, where:*
    - *$q, q' \in Q$ are the source and destination locations,*
    - *$a \in Act$ is the action of the transition,*
    - *$\lambda \subseteq C$ is the set of clocks to reset to zero,*
    - *δ is a clock constraint over C. δ is defined inductively by*

    $$\delta := x\#c \mid \delta_1 \wedge \delta_2,$$

    where c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$.

◇

Figure 2.8 presents an example of a timed automaton that models the same scenario represented in Figure 2.6. Considering timed automata, it is assumed all clocks are reset to

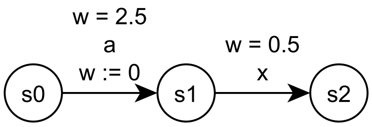


Figure 2.8: TA Example

zero at the beginning. In Figure 2.8, there is a single clock w and the notation $w := 0$ means the action of resetting the clock w. Similarly, the notations $w = 2.5$ and $w = 0.5$ represent the clock constraints associated with the transitions. The automaton starts in location s_0 and, once the transition is enabled (i.e. $w = 2.5$), the action a is executed, reaching the location s_1. The clock is reset to zero along with this transition. The time elapsed since the occurrence of the action a is shown by the value of clock w. The transition from location s_1 to location s_2 is enabled only if this value is equal to 0.5, where the action x is immediately executed.

The semantics of TA can be defined in terms of an infinite TLTS (Definition 2.18). Let the function $v : C \rightarrow \mathbb{R}^{\geq 0}$ denote a clock valuation v.

Definition 2.18 (TLTS semantics of a TA)**.** *The semantics of a TA $A = \langle Q, q_0, Act, C, E \rangle$ is a TLTS $L_A = \langle S, s_0, Act, T \rangle$, defined as follows:*

- *$S = Q \times (C \rightarrow \mathbb{R}^{\geq 0})$ is the set of states of the form $s = (q, v)$ where $q \in Q$ is a location and v is a clock valuation;*
- *$s_0 = (q_0, \vec{0})$ is the initial state, where $\vec{0}$ is the valuation assigning 0 to every clock in C;*
- *$Act = A \cup D$ is the set of actions, where A is the set of discrete actions and $D = \{d \mid d \in \mathbb{R}^{\geq 0}\}$ is the set of time-elapsing actions;*
- *T is the transition relation defined as follows: (1) transitions with discrete actions are of the form $(q, v) \xrightarrow{a} (q', v')$, where $a \in Act$ and there is a transition $\langle q, q', a, \lambda, \delta \rangle$, such that v satisfies δ and v' is obtained by resetting to zero all clocks in λ; (2) transitions with time-elapsing actions are of the form $(q, v) \xrightarrow{d} (q, v + d)$ for all $d \in \mathbb{R}^{\geq 0}$ such that $v \models \delta$ and $v + d \models \delta$.*

◇

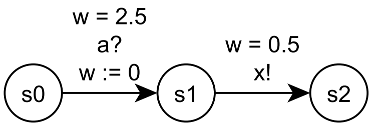


Figure 2.9: TAIO Example

Timed Automata with Inputs and Outputs (TAIO) is an extension of TA where the set of actions Act is partitioned in two disjoint sets: a set of input actions and a set of output actions.

Definition 2.19 (TAIO)**.** *A TAIO is a timed automata $\langle Q, q_0, Act, C, E\rangle$ where $Act = Act_I \cup Act_O$, such that Act_I is a finite set of input actions and Act_O is a finite set of output actions. Moreover, $Act_I \cap Act_O = \emptyset$.* ◇

The semantics of TAIO can be defined in terms of an infinite TIOLTS in a similar manner as described for TA. Figure 2.9 presents an example of a TAIO. This example represents the scenario of Figure 2.8 and it states that the system must receive the input a exactly with 2.5 units of time. Finally, the system must output b exactly with half of one unit of time.

As the semantics of TA and TAIO is defined in terms of infinite timed labelled transition systems, both verification and testing techniques must deal with large sets of states which may lead to the state space explosion problem. In this case, it is important to have an efficient symbolic representation of the state space. One of the most efficient representations is based on the notion of zone [42; 58; 122; 123]. A zone represents the maximal set of clock valuations that satisfy a constraint.

The analysis of the state space using this symbolic representation requires some operations such as:

- The future of a zone Z, defined by $\vec{Z} = \{z + d \mid z \in Z,\ d \in \mathbb{R}^{\geq 0}\}$;
- The intersection of two zones Z and Z', defined by $Z \cap Z' = \{z \mid z \in Z,\ z \in Z'\}$;
- The reset to zero $\lambda \subseteq C$ of Z, defined by $[\lambda \leftarrow 0]Z = \{[\lambda \leftarrow o]z \mid z \in Z\}$.

All these operations are graphically illustrated in Figures 2.10, 2.11, and 2.12, respectively.

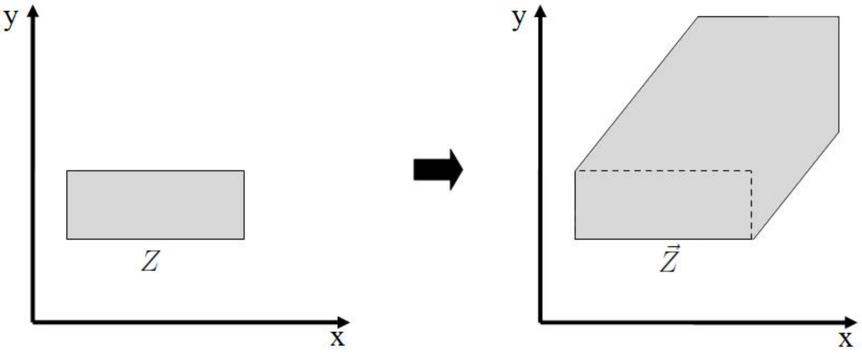


Figure 2.10: Example of the Future Operation

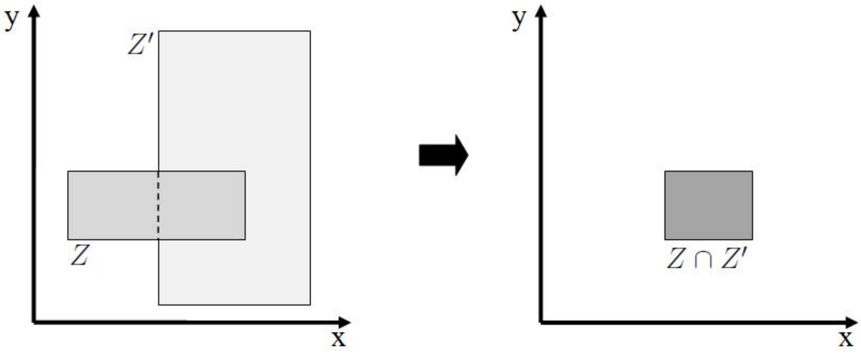


Figure 2.11: Example of the Intersection Operation

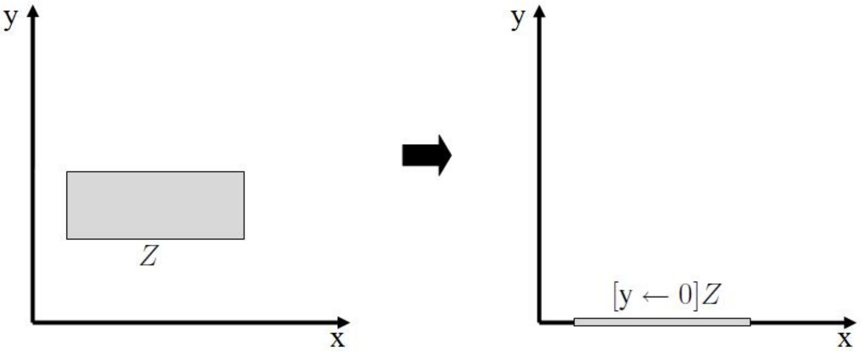


Figure 2.12: Example of the Reset to Zero Operation

2.2.4 Testing of Real-Time Systems

The increasing use of real-time systems, in most different contexts, has been demanding investments in order to increase the reliability and integrity of such systems. Some research efforts have been expended in devising techniques such as model checking [13; 31; 54], where the correctness of models is verified in an automated and accurate manner. However, if the same rigour is not applied to the test of the implementation, a gap is created between these processes, allowing the presence of defects in the implementation even if the model had been successfully verified.

Since research in the real-time software testing field is very recent, the developed techniques and tools are still immature and difficult to use in practice. Thus, one of the challenges today is the search for methods, techniques and tools to support the test of systems with time restrictions. Nevertheless, real-time systems have several distinguishing characteristics that may need to be taken into account during the testing process, leading to the most difficult challenges in software testing [121].

The test of real-time systems is more difficult than the test of non-real-time systems because the correct behaviour of the former depends not only on the correct results but also when they are emitted. Thus, it is essential to develop techniques to interact with the system: (1) by providing inputs at the correct time and (2) by observing and evaluating if the generated results are correct in terms of integrity and timing.

Real-time systems are usually composed of parallel activities. So, the models must represent such parallelism between many elements and allow ways of communication between these elements. As previously said, RTS are extremely related to events that often occur in terms of interruption signals from the arrival of data, ticking of a hardware clock, or an error alarm. To provide an effective solution for testing, it is crucial to define models capable of representing these asynchronous events. In addition, the model has to be composable, allowing events to be combined at different points of possibly different flows of execution. Research in this direction is practically nonexistent and some approaches only consider interruptions in a non-real-time context [7; 8; 9; 12; 29; 39].

Another hard activity of testing real-time systems is the test execution. Considering an environment composed of several processes executing at the same time with synchronous and asynchronous events, it is very difficult to have control of the whole environment. Current work in the literature consider so many hypotheses related to the environment and the system under test that the results sometimes are not useful in practice. It is needed testing techniques and theories that allow one to have interesting conclusions without having the environment fully controllable.

A pass verdict is emitted when the system produces an acceptable result on time. The addition of time in the validation of RTS, the unpredictability of events, and the absence of controllability during the test execution lead to a harder oracle problem. This happens due to the fact that it is very difficult to have total control of an environment with parallel activities during the test execution. Moreover, in practice, the time cannot be controlled and it is treated in different abstraction levels by the tester and the specification. In this case, test hypotheses must be defined in order to achieve valid verdicts. To the best of our knowledge, approaches in the direction of solving the oracle problem considering the context of real-time systems are practically nonexistent.

2.3 Concluding Remarks

This chapter presented the theoretical basis necessary for the understanding of this work. At the beginning, the main concepts related to software testing were presented, for instance test cases, oracles, approaches to identify test cases, conformance testing, and testing techniques like model-based testing, property oriented testing, and symbolic testing.

With respect to the real-time systems context, we described what a real-time system is and discussed several characteristics like how the time can be modelled, the types of existing events, some classical notations used to model RTS, and the difficulties to test real-time systems.

Chapter 3

Interruption Testing of Reactive Systems

In order to provide an effective solution for interruption testing, it is crucial to define a model capable of representing such interruptions and, consequently, make the automatic test case generation process possible. In addition, the model has to be composable, allowing interruptions to be combined at different points of possibly different flows of execution. Moreover, due to the large amount of possible test cases, selection strategies need to be applied to reduce the size of test suites. Furthermore, the test execution environment should be carefully considered so that execution requirements and constraints are properly identified and handled.

This chapter presents an approach to conformance testing of reactive systems with interruptions that covers modelling (devoted to testing), generation and selection of sound test cases [9]. The model adopted is named *Annotated Labelled Transition System* (ALTS). This kind of Labelled Transition System (LTS) has special descriptions inserted into the model in order to make the test case generation process feasible. LTSs are good models for functional testing because all needed information is the observable interactions between applications and environment and between applications. Also, they are the underlying formalism of most formal notations for reactive applications. The proposed model is implemented by the LTS-BT tool [29] and a case study is performed to illustrate the benefits of the strategy when compared to manual selection.

Interruption testing was investigated in [6] considering the mobile phone applications context, but only an operational strategy was proposed. This chapter presents a formalised approach for reactive systems in general. Furthermore, the work presented in [9] and described in this chapter extends the work presented in [8], covering the following aspects: (1) algorithms defined to translate high level specifications into ALTS models are presented; (2) the case study is presented in more details and results are more thoroughly discussed.

3.1 Context

In general, the test process in the context of this chapter starts with a specification of the System Under Test (SUT) and interruptions. Given high level specifications, an ALTS model is automatically generated. Finally, the ALTS model is combined with test purposes for interruption test case generation. The interruption test process uses test purposes in order to test at specific points of interest. A general view of this test process is presented in Figure 3.1. This process considers the test architecture presented in Figure 3.2. In this test architecture, two elements are important: the SUT and the TESTER. The SUT is composed of the main application and interruptions allowed during the test process. The environment is assumed to be fully controllable by the TESTER, thus, during test execution the TESTER has total control of the interruptions, deciding when they start and finish.

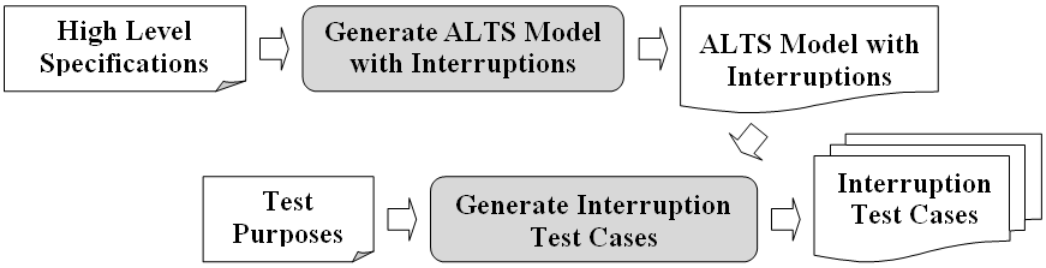

High Level Specifications
Generate ALTS Model with Interruptions
ALTS Model with Interruptions
Test Purposes
Generate Interruption Test Cases
Interruption Test Cases

Figure 3.1: Interruption Test Process

The SUT is specified as use cases using a controlled natural language [26; 84; 115]. An example of a use case of a mobile phone application is shown in Figure 3.3. This represents the behaviour of removing a message from inbox. A use case must have a main flow and can have some alternative flows. The flows are described through steps that include a user action and the respective system response. For instance, the step “4M" has the selection of the “Remove" option, and the respective system response is to show an alert saying that the message was removed.

Besides the actor action and the system response, each step has a condition (System State)

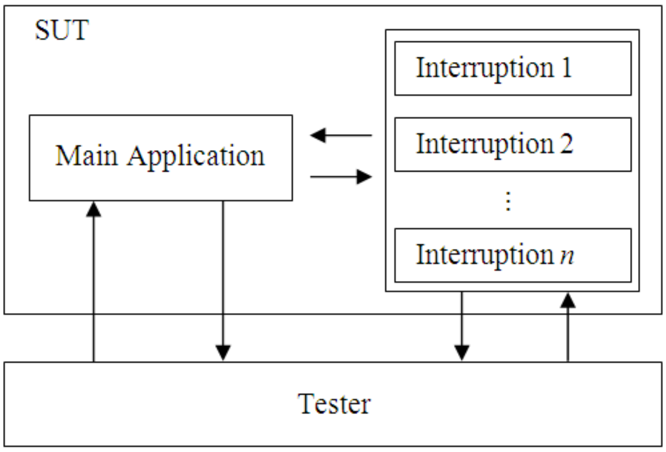
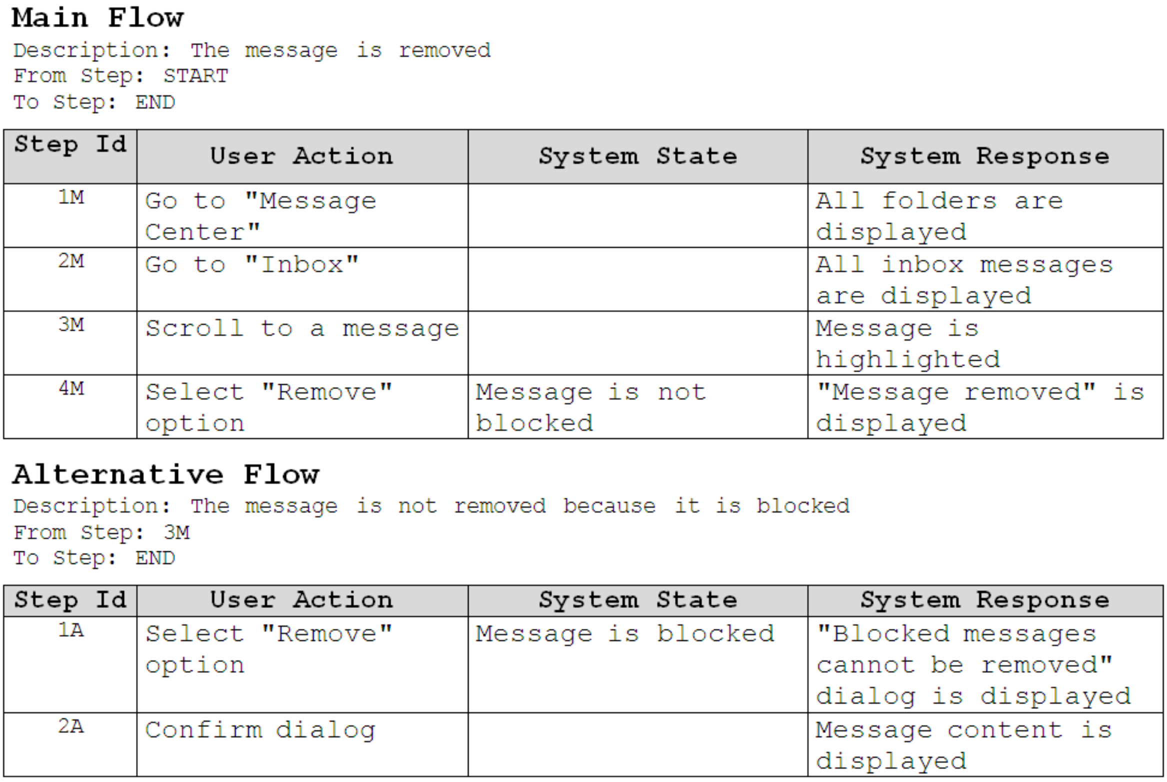


Figure 3.2: Test Architecture



Main Flow

Description: The message is removed
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Go to "Message Center"		All folders are displayed
2M	Go to "Inbox"		All inbox messages are displayed
3M	Scroll to a message		Message is highlighted
4M	Select "Remove" option	Message is not blocked	"Message removed" is displayed

Alternative Flow

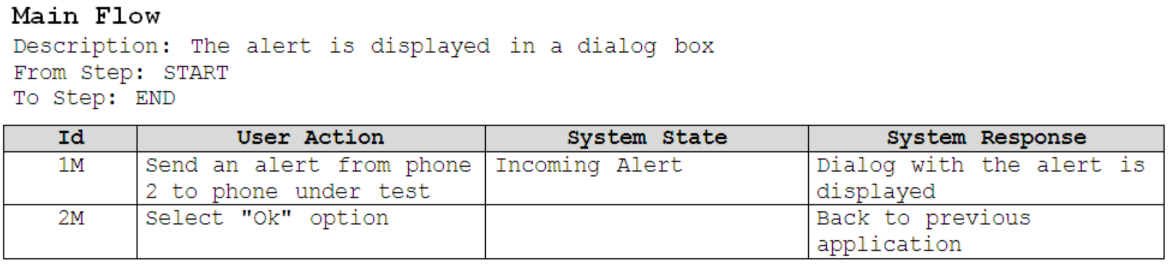
Description: The message is not removed because it is blocked
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Select "Remove" option	Message is blocked	"Blocked messages cannot be removed" dialog is displayed
2A	Confirm dialog		Message content is displayed

Figure 3.3: *Remove Message* Specification

that determines if the system response will happen or not. If the condition is not satisfied, an alternative flow must be specified. As an example, the step “4M" of the main flow has one alternative flow (steps “1A" and “2A").

Considering the specifications of interruptions, the idea is to specify an interruption in the same way by using the same use case template that is used to specify a simple behaviour of the SUT [39]. For instance, Figure 3.4 presents the behaviour of an incoming alert interruption. This interruption specifies the arrival of a new kind of text messages where the text appears to the user inside a dialog box.



Main Flow
Description: The alert is displayed in a dialog box
From Step: START
To Step: END

Id	User Action	System State	System Response
1M	Send an alert from phone 2 to phone under test	Incoming Alert	Dialog with the alert is displayed
2M	Select "Ok" option		Back to previous application

Figure 3.4: *Incoming Alert Interruption* Specification

Once the interruption flow is specified, we assume that it can be executed at any time of another use case execution, that is, between any step of another use case. With this specification strategy, interruption behaviours are defined in a simple manner and all points where an interruption can occur do not need to be explicitly specified.

3.2 Interruption Model

This section presents the proposed ALTS model structure capable of representing interruptions. Firstly, interruptions are represented with IOLTS models (Definition 2.2) to illustrate the challenges and the desired semantics for ALTS models. Secondly, ALTS models are defined, their structure is illustrated by an application in the mobile phone domain, and their semantics are defined showing how an ALTS can be converted into an IOLTS. Finally, the notion of conformance considered is discussed.

3.2.1 Representing Interruptions with IOLTS Models

Considering the conformance testing approach, LTS is one of the most used formalisms. Basically, LTS models are represented by graphs where the nodes are the possible system states and the edges represent the transition between these states through occurrence of actions. LTSs can be used for modelling the behaviour of systems such as specifications, implementations, and tests, and it serves as a semantic model for several formal languages such as CCS and CSP [116; 117].

Particularly in the case of reactive systems, the underlying model should represent the interaction of the system with its environment by distinguishing between inputs and outputs. In this case, IOLTS models are used. Figure 3.5 shows an example of an IOLTS. An input event is defined using the symbol "?" followed by the event name and an output event is defined using the symbol "!" followed by the event name.

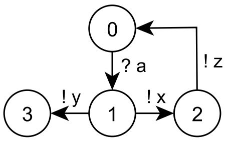


Figure 3.5: Simple IOLTS

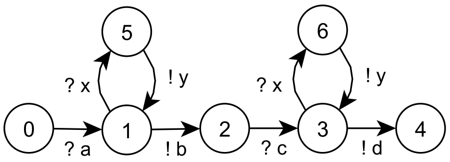


Figure 3.6: Modelling Interruptions Using IOLTS

It is possible to model interruptions using an IOLTS. For this, each possibility of interruption needs to have a specific set of states, implying that interruption flows must be duplicated. Figure 3.6 shows an example of how to model interruptions using IOLTS. Nodes from 0 to 4 are related to a behaviour that can be interrupted by another behaviour at nodes 1 and 3. State 5 represents the possibility of interruption at node 1 and state 6 the possibility of interruption at node 3. Note that nodes 5 and 6 represent the same interruption behaviour.

The replication of the interruption model is due to the semantics of the behaviour. Suppose that only one state had been used to represent the interruption behaviour, then it would not be possible to associate a unique next state to the end of the interruption execution. After an interruption execution, the flow needs to continue from the same point where the interruption had started.

3.2.2 Annotated Labelled Transition Systems

ALTSs are capable of representing interruptions in a more compact way, following the same semantics presented in the previous subsection. This new kind of LTS follows the same classical LTS definition. The difference is that each label is associated with a description. This new description inserted into the model is called an annotation. Before defining an ALTS, a definition of a Generic Annotated Labelled Transition Systems (GALTS) is presented.

Definition 3.1 (GALTS)**.** *A GALTS is a 5-tuple $\langle Q, A, L, q_0, T\rangle$, where:*

- *Q is a countable, non-empty set of locations;*
- *A is a countable, non-empty set of annotations;*
- *L is a countable, non-empty set of labels;*
- *$q_0 \in Q$ is an initial location;*
- *T is a set of transitions. Each transition consists of:*
  - *a location $q \in Q$, called the origin of the transition;*
  - *an annotation $a \in A$, called the annotation of the transition;*
  - *a label $l \in L$, called the label of the transition;*
  - *a location $q' \in Q$, called the destination of the transition.*

◇

As said before, each label has an associated description (annotation). So in the GALTS definition (Definition 3.1) we have a set A that contains the possible descriptions of the labels. This set can be instantiated according to the information to be modelled or the context where the model will be used. In this work, the focus is on a model to support the test process, mainly a model capable of representing interruptions efficiently. Thus, a more specific GALTS is defined where the set A of annotations has predefined elements.

Definition 3.2 (ALTS)**.** *An ALTS is a 5-tuple $\langle Q, A, L, q_0, T\rangle$, where:*

- *Q is a countable, non-empty set of locations;*

- *A = {steps, conditions, expectedResults, beginInterruption_X, endInterruption_X} is the set of annotations;*
- *L is a countable, non-empty set of labels;*
- *$q_0 \in Q$ is an initial location;*
- *T is a set of transitions. Each transition consists of:*
    - *a location $q \in Q$, called the origin of the transition;*
    - *an annotation $a \in A$, called the annotation of the transition;*
    - *a label $l \in L$, called the label of the transition;*
    - *a location $q' \in Q$, called the destination of the transition.*

$\diamond$

These annotations were chosen with the following specific goals: (1) guide the test case generation process, by making the focus on particular interruptions easier; (2) make it possible for interruption models to be plugged and unplugged without interfering with the main model; (3) guide test case documentation; (4) make it possible for conditions to be associated with actions; (5) indicate points where interruptions can be reasonably observed externally.

The annotation *steps* is associated with a label $l \in L$ (we write $[steps]l$) to indicate that l is an input action. When a label $l \in L$ represents a condition associated with an input action, we use the annotation *conditions* and write $[conditions]l$. The expected results are indicated through *expectedResults* annotation ($[expectedResults]l$). Two other annotations are used to indicate the start and the end of an interruption and they are considered as special kinds of input actions and expected results, respectively. So the labels in *L* represent the observable actions (input or output actions) or some condition associated with these actions.

Let $W = \langle Q, A, L, q_0, T \rangle$ be an ALTS. We write $q \overset{[a]l}{\rightarrow} q'$ for $(q, a, l, q') \in T$ and $q \overset{[a]l}{\rightarrow}$ for $\exists q' : q \overset{[a]l}{\rightarrow} q'$. An ALTS can be defined by its initial location, then we write W $\rightarrow$ for $q_0 \rightarrow$. Depending on the associated annotation, the labels can be classified as input actions, output actions, and conditions. Thus, let $L = L_I \cup L_O \cup L_C$, where L_I is the set of input actions, L_O is the set of output actions, and L_C is the set of conditions. Let $a_{(i)} \in A$ be some annotations, $\omega_{(i)} \in L$ be some labels, $\sigma \in ([A]L)^*$ a sequence of labels with their respective annotations, and q, $q' \in Q$ some locations.

Let $\Omega(q) \triangleq \{[a]\omega \mid a \in A, \omega \in L, q \xrightarrow{[a]\omega}\}$ be the set of actions reachable from q. Also, let $Out(q) \triangleq \Omega(q) \cap [A \setminus \{steps, conditions, beginInterruption\_X\}]L_O$ be the set of outputs reachable from q. The definition of $Out(q)$ can be extended for sets of locations: for $P \subseteq Q$ we have $Out(P) \triangleq \bigcup_{q \in P} Out(q)$. Denote $q \xrightarrow{[a_1]\omega_1 \ldots [a_n]\omega_n} q' \triangleq \exists q_0, \ldots, q_n : q = q_0 \xrightarrow{[a_1]\omega_1} q_1 \xrightarrow{[a_2]\omega_2} \ldots \xrightarrow{[a_n]\omega_n} q_n = q'$. The set q *after* $\sigma \triangleq \{q' \in Q \mid q \xrightarrow{\sigma} q'\}$ is the set of locations reachable from q, and P *after* $\sigma \triangleq \bigcup_{q \in P} q$ *after* σ is the set of locations reachable from the set P. *Traces*$(q) \triangleq \{\sigma \in ([A]L)^* \mid q \xrightarrow{\sigma}\}$ describes the sequences of labels with their respective annotations reachable from q. Considering the sequences of labels and annotations reachable from the initial location of an ALTS W, we define *Traces*$(W) \triangleq$ *Traces*(q_0).

Considering our running example presented in Section 3.1, Figure 3.7 presents an ALTS model that represents the behaviour of removing a message from inbox. This application is specified by the use case shown in Figure 3.3. The model where interruptions can occur will be illustrated with the scenario where that feature specified by the use case shown in Figure 3.3 can be interrupted at some points by the Incoming Alert interruption (interruption specified by the use case presented in Figure 3.4). The scenario described above is presented in Figure 3.8. Note that locations from 0 to 13 are related to the remove message behaviour (Figure 3.3), and locations from 14 to 17 are related to the Incoming Alert interruption (Figure 3.4).

From Figure 3.8, the interruption model is connected to the feature that can be interrupted

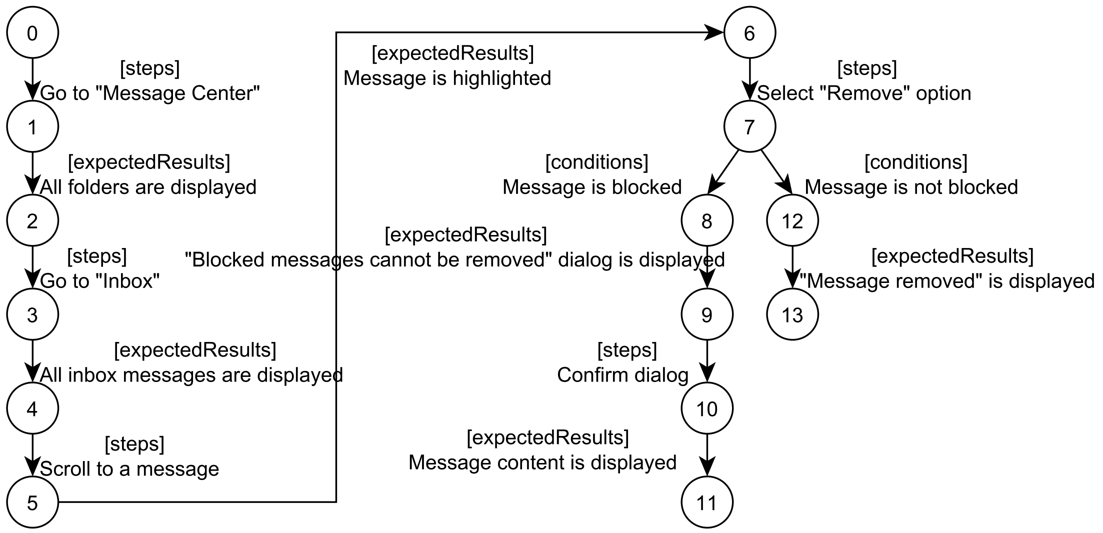


Figure 3.7: *Remove Message* behaviour

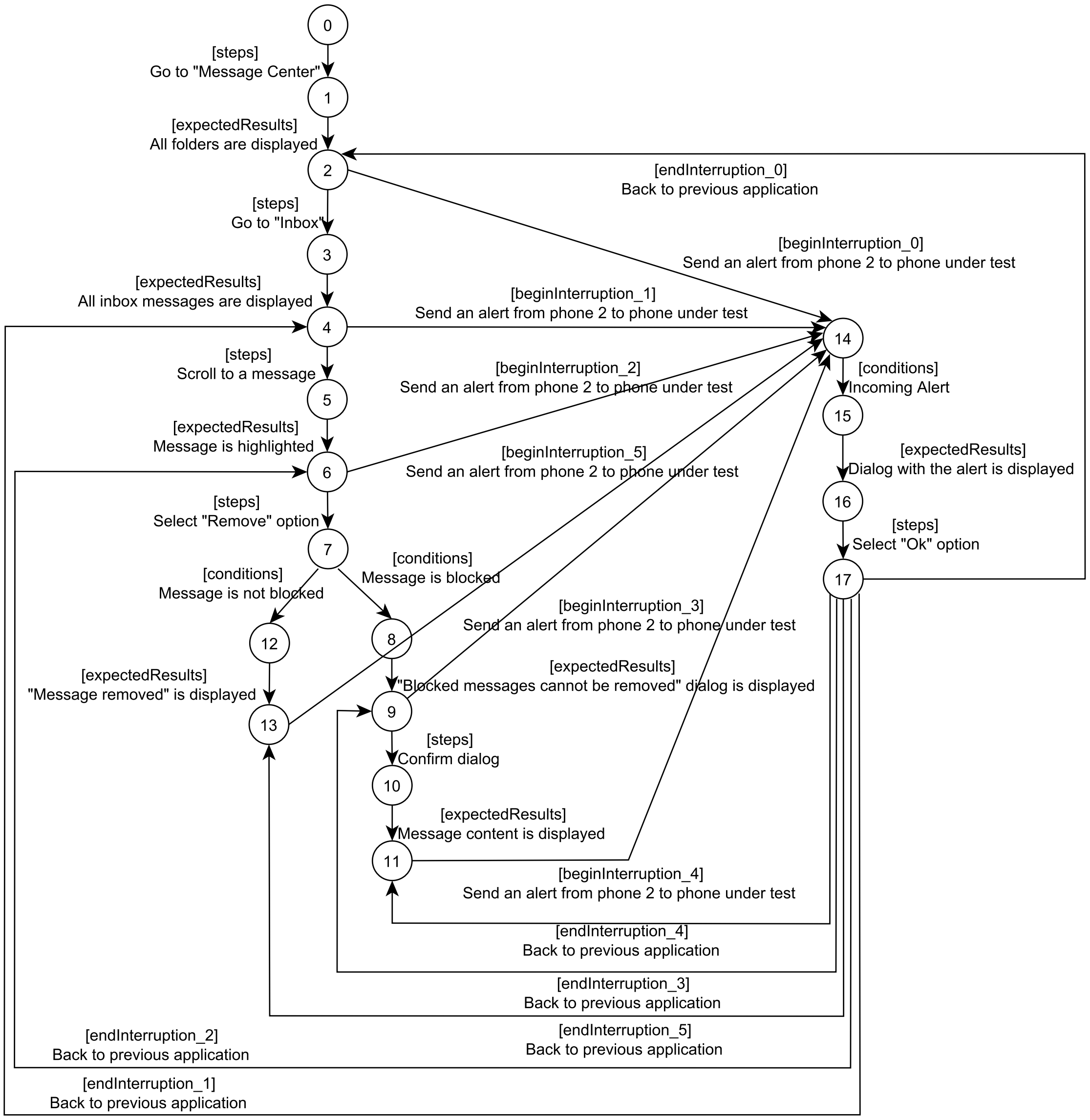

0
[steps]
Go to "Message Center"
1
[expectedResults]
All folders are displayed
2
[endInterruption_0]
Back to previous application
[steps]
Go to "Inbox"
3
[beginInterruption_0]
Send an alert from phone 2 to phone under test
[expectedResults]
All inbox messages are displayed
[beginInterruption_1]
Send an alert from phone 2 to phone under test
4
14
[steps]
Scroll to a message
[beginInterruption_2]
Send an alert from phone 2 to phone under test
[conditions]
Incoming Alert
5
15
[expectedResults]
Message is highlighted
[expectedResults]
Dialog with the alert is displayed
6
[beginInterruption_5]
Send an alert from phone 2 to phone under test
16
[steps]
Select "Remove" option
[steps]
Select "Ok" option
7
[conditions]
Message is blocked
[conditions]
Message is not blocked
17
[beginInterruption_3]
Send an alert from phone 2 to phone under test
12
8
[expectedResults]
"Blocked messages cannot be removed" dialog is displayed
[expectedResults]
"Message removed" is displayed
13
9
[steps]
Confirm dialog
10
[expectedResults]
Message content is displayed
11
[beginInterruption_4]
Send an alert from phone 2 to phone under test
[endInterruption_4]
Back to previous application
[endInterruption_3]
Back to previous application
[endInterruption_2]
Back to previous application
[endInterruption_5]
Back to previous application
[endInterruption_1]
Back to previous application

Figure 3.8: *Remove Message* behaviour with Interruptions

(the main flow) using two new annotations: *beginInterruption_X* and *endInterruption_X*, where *X* is a counter. These annotations are used to indicate where the main flow has been interrupted. Also, they are needed to represent the behaviour where the main flow continues its execution from the same point where it had been interrupted. For instance, if an interruption begins with the *beginInterruption_0* annotation it must finish with *endInterruption_0*.

One of the main advantages of using the Annotated LTS is that we can add the same interruption behaviour to many different points only by manipulating the two new annotations (*beginInterruption_X* and *endInterruption_X*). Thus, we can represent interruptions in a more compact way than standard LTS, while preserving the same efficiency and precision in test generation (this is discussed in Section 3.4).

Considering time as being continuous, an interruption can occur at infinite points during the system execution. But considering the tester's point of view, each possibility of interruption can only be observed after each system response. This happens due to the fact that it is impractical to reproduce a scenario where an interruption occurs between an input action and the system response, mainly when tests are manually executed. It is important to remark that this is a limitation of the test process in general and not of the proposal presented in this chapter. Thus, the intention is to represent only interruptions that occur immediately after the system responses. In this case, Figure 3.8 represents all possibilities of interruption from the tester's point of view.

Note that, as we are considering an LTS model for testing, only functionalities to be tested are specified. Thus, we have a partial behavioural model. From the tester's point of view, only the specified behaviour is observed, and with this, all other behaviours are not observed during the test, including other possible interruptions. We are assuming that the test execution environment is controlled by the tester, that is, an interruption only occurs when the tester wishes that it occur.

In practice, this interruption model should not be written by hand because it is tiresome and not cost-effective. It must be generated directly from abstract specifications. The ALTS model presented in this section is automatically generated from those use case templates described in Section 3.1 by the LTS-BT tool [29]. This tool is described in more details in the next section.

The semantics of ALTS models can be defined in terms of IOLTS models. Basically, locations in the ALTS model are states in the IOLTS plus additional states that are created to replicate the interruption behaviour. Labels annotated with *steps*, *conditions*, *beginInterruption_X* are input actions in the IOLTS, whereas labels annotated with *expectedResults* and *endInterruption_X* are output actions in the IOLTS. The transition function is incremented by the replication of interruption behaviour by considering the new states added. The most non-trivial element in this association is the condition transition. The reason to map them to input action is that they often represent in test models different paths of execution of the application. When a condition is associated with a test case, this often means that the tester will need to properly set up the application so that a particular flow of execution can be tested.

Therefore, in test case documentation, they are often promoted to an initial condition that will demand an input set up information.

3.2.3 Testing Conformance

This work considers a testing theory that is based on the notions of *specification*, *implementation*, and a *conformance relation* between them [118]. The specification of a reactive system with interruption can be written in any notation that can be transformed into an ALTS model. But this work considers only use case templates or an ALTS that respects the constraints on the use of labels defined in Subsection 3.2.2. The implementation can be any computer system that can be interrupted at any time and can be modelled as an ALTS. Moreover, it is assumed all interruptions to be controllable and implementations to be input-enabled, that is,

$$\forall q \in Q, a \in A \setminus \{conditions, expectedResults, endInterruption\_X\}, \forall \omega \in L_I, q \overset{[a]\omega}{\rightarrow}.$$

As discussed in Section 3.1, the tester needs full control of the test environment in order to achieve valid verdicts during the test execution process. Thus, the implementation under test must respond to all stimuli of the tester leading to require, at least during the test execution, that the implementation is input-enabled. This is a usual assumption of testing techniques (e.g. [62; 116; 117]) that can be achieved by appropriate control mechanisms. This work considers a conformance relation based on a simplification of the **ioco** relation defined by Tretmans in [117]. The **conf** relation (presented in Definition 3.3) does not take internal actions and quiescence into account.

Definition 3.3 (conf)**.** *Let the specification S be an ALTS and SUT be an input-enabled ALTS:*

$$\mathit{SUT}\ \boldsymbol{conf}\ S \triangleq \forall \sigma \in \text{Traces}(S),\ Out(\mathit{SUT} \text{ after } \sigma) \subseteq Out(S \text{ after } \sigma).$$

◇

It is important to mention that traces of ALTSs are restricted to paths in which the returns from interruptions go to the correct locations according to the ALTS semantics presented in the end of Section 3.2.2. Informally, an implementation conforms to a specification for **conf** if for all traces of the specification, the set of output actions of the implementation in each location is contained in the set of output actions of the specification. This implementation relation is similar to the one considered by the TGV tool [62].

3.3 Interruption Test Case Generation and Selection

This section presents the algorithms developed in order to automate the test process described in Section 3.1. Firstly, algorithms that translate use case templates to ALTS models are presented. After that, an algorithm that generates interruption test cases is shown. Finally, an interruption test case selection strategy based on test purposes is presented.

3.3.1 ALTS Model Generation from Use Case Templates

This subsection presents a strategy for translating use case templates into ALTS models from which test cases can be generated. The general translation procedure is shown in Algorithm 3.1. This procedure is a variation of the one presented in [93] that focus on individual features only. Basically:

- Each template of the use case, starting from the main flow one, is processed sequentially and, from each step, locations and transitions are created in the target ALTS according to the order of steps defined in the template. This is controlled by the two *for* loops;

- *currentLocation* represents the location from which transitions are created for the current step. This is either: (1) the last location created in case the *From Step* field is defined as *START* or this is the first location; or (2) the last location of a given step (defined in the *From Step* field) of another template;

- *From Step* and *To Step* guide the connection of each trace created by each of the templates;

- *User Action*, *System State*, and *System Response* become transitions that are associated with *steps*, *conditions*, and *expectedResults* annotations, respectively;

- Locations are created as new transitions that need to be added. These are incrementally numbered from 0. Locations and transitions are created by the add operation. But locations already created can be reused when connecting the traces of new templates. When this is possible, the *addToStep* (*To Step* is different from *END*) and *addFromStep* (*From Step* is different from *START*) are used instead;

- Duplicated transitions from the same location are also avoided. This can happen when the same steps are possible but with different conditions.

Algorithm 3.1 is based on two loops in order to process all steps of all templates. In practice, all steps of all templates are processed only one time. Thus, the running time of Algorithm 3.1 using the asymptotic notation is $O(|\text{STEPS}|)$, where | STEPS | represents the sum of all steps of all templates to be processed.

After the generation of the ALTS models from use case templates, the model of the main application and the model of the interruption must be connected. Algorithm 3.2 is responsible for connecting them. Basically:

- The procedure uses a Depth-First Search (DFS) strategy for traversing all locations of the main application, that is, the application to be interrupted;
- As the proposed strategy only considers interruptions to be possible after expected results, the only possible points of interruptions are exactly after the transitions with the *expectedResults* annotation;
- When a transition with an *expectedResults* annotation is found in the main application model, a new transition (with the *beginInterruption_X* annotation) is added to the first location of the interruption model. And, from each final location of the interruption model, a new transition (with the *endInterruption_X* annotation) is added for connecting this model with the main application model;
- *Search* is a procedure responsible for marking a location (its second parameter) as a visited location and putting all adjacent locations in a list (its first parameter) to be processed.

The running time of Algorithm 3.2 using the asymptotic notation is $O(\mid Q \mid \cdot \mid F \mid)$, where $|Q|$ is the number of locations of the main application and $|F|$ is the number of final locations of the interruption model.

Algorithm 3.1: Procedure that Translates Use Case Templates to an ALTS

```
UseModel UseCase2ALTS(Collection Templates) {
  UseModel alts := new UseModel();
  for each template in Templates {
    if (template.getFromStep() $\neq$ START) {
      // location after expected results of from step
      currentLocation := alts.getFinalLocation(template.getFromStep());
    } else {
      // associates a location to the step that creates it
      currentLocation := new Location(template.getFirstStep());
    }
    for each step in template {
      Transition steps := step.getUserAction();
      Transition conditions := step.getSystemState();
      Transition expectedResults := step.getSystemResponse();
      if (steps in alts) {
        currentLocation := alts.getLocationAfter(steps);
      } else if (step = template.getFirstStep() AND
          template.getFromStep() $\neq$ START) {
        // Avoid duplicating a steps trans. from the same location
        currentLocation := alts.addFromStep(currentLocation, steps,
            template.getFromStep());
      } else {
        // Create a new location for adding the new transition
        currentLocation := alts.add(currentLocation, steps);
      }
      if (conditions $\neq$ emptyCondition) {
        currentLocation := alts.add(currentLocation, conditions);
      }
      if (step = template.getLastStep() AND template.getToStep() $\neq$ END) {
        // The target loc. for the system resp. trans. is already created
        currentLocation := alts.addToStep(currentLocation,
            expectedResults, template.getToStep());
      } else {
        currentLocation := alts.add(currentLocation, expectedResults);
      }
    }
  }
  return alts;
}
```

Algorithm 3.2: Procedure that Combines the Main Application Model with an Interruption Model

```
CombineALTSModels(UseModel mainApplication, UseModel intModel, Integer intCode) {
  List list := $\emptyset$; // List of transitions to be visited
  Collection finalLocations := intModel.getFinalLocations();
  search(list, mainApplication.getRootLocation());
  while (list $\neq$ $\emptyset$) {
    transition := list.remove();
    targetLocation := transition.getToLocation();
    if (transition.isExpectedResultsTransition()) {
      targetLocation.addBeginInterruptionTransition(intModel.getRootLocation(), intCode);
      for each location in finalLocations {
        location.addEndInterruptionTransition(targetLocation, intCode);
      }
      intCode++;
    }
    if (targetLocation is not visited)
      search(list, edge.getToLocation());
    }
  }
}
```

3.3.2 Interruption Test Case Generation

This subsection describes the interruption test case algorithm developed to extract test cases from ALTS models. A test case generated from an ALTS is defined as follows.

Definition 3.4 (Test Case). *A test case is an ALTS* $TC = \langle Q^{TC}, A^{TC}, L^{TC}, q_0^{TC}, T^{TC} \rangle$. *The set of annotations is the same as the specification (*$A^{TC} = A^{S}$*) and the set of labels is* $L^{TC} = L_I^{TC} \cup L_O^{TC} \cup L_C^{TC}$, *where* $L_I^{TC} \subseteq L_O^{SUT}$ *(outputs of the SUT are the inputs of the TC),* $L_O^{TC} \subseteq L_I^{SUT}$ *(TC emits only inputs allowed by the SUT), and* $L_C^{TC} \subseteq L_C^{S}$ *(the conditions are the same specified by the specification).* ◇

Test cases can be obtained from ALTS models, using the DFS method, by traversing the

Algorithm 3.3: Test Case Generation Algorithm

```
Decompose(Location loc, Path path, Integer intCode) {
  if (loc.isLeaf() OR (loc.isRoot() AND path $\neq \emptyset$)) {
    // End of a path
    recordTestCase(path);
    return;
  }
  for each descendent in loc.getAdjacencies() {
    edge := getEdgeBetween(loc, descendent)
    if (edge.isBeginInterruption()) {
      intCode := edge.getIntCode();
    }
    if ((edge.getIntCode() = $-1$ AND edge $\notin$ path) OR (intCode >= 0 AND edge.getIntCode() = intCode)) {
      path.add(edge);
      if (edge.isEndInterruption()) {
        intCode := $-1$;
      }
      Decompose(descendent, path, intCode);
    } else if (edge.getIntCode() = intCode) {
      recordTestCase(path);
    }
  }
  return;
}
```

ALTS starting from the initial location (see Algorithm 3.3). As a general coverage criterion, all transitions need to be covered, i.e., all transitions of the ALTS model are visited at least once. As Algorithm 3.3 is based on DFS, its running time using the asymptotic notation is $O(|Q| + |T|)$, where $|Q|$ is the number of locations and $|T|$ is the number of transitions of the ALTS model.

Algorithm 3.3 requires three parameters: *loc*, a location of the model, indicating the current one during execution; *path*, a set of transitions from the model, indicating the path visited during the processing; and *intCode*, the interruption code, indicating that a given interruption is being processed.

The extraction is started from the root (the initial location of the ALTS model), verifying if the current location indicates the end of a path in the model, indicating that the test case has been extracted. In this case, it needs to be recorded. If the current location does not indicate the end of a path, then each of its descendants is visited through the depth-first search strategy.

To visit each of its descendants, the edge between the current location and its descendant is analysed. The search proceeds only if (Algorithm 3.3, Line 12): (i) the edge does not belong to the current analysed path, i.e., the edge has not already been "visited" (note that when the algorithm is processing the main application, the value of *intCode* is -1); or (ii) if it is an edge from an interruption behaviour (an edge with the *endInterruption_X* label). This precaution is necessary because after the interruption, the extraction process in the ALTS comes back to previous location (the last location of the main application before the interruption), therefore being possible to pass through the same interruption, in different parts of the model, and constraining that would cause inconsistency.

Due to these conditions, two scenarios are encountered: (1) Conditions (i) and (ii) are not satisfied: The search stops, recording the entire path as a test case avoiding loops in the main application and finishing an interruption with the correct *endInterruption_X* transition. In this case, the recursion step of the algorithm returns to the next branch that needs to be analysed, continuing the algorithm; (2) Condition (i) or (ii) is satisfied: The edge between the location and its descendent is added to the test case and the algorithm continues until it finds the end of the path, which happens when either a leaf in the graph or an edge going back to the root of the model are found.

These constraints over the extraction, when using the depth-first search approach, are required to avoid an explosion of paths during the test case extraction caused by loops in the ALTS model. This may reduce the number of extracted test cases, but without those constraints, the number of paths extracted becomes unfeasible, while most of them may be obtained by combining the extracted test cases. Also, from a functional testing point of view, in practice these excluded paths generally add redundancy to the test suite, that is, they do not generally add test cases that would uncover escaped faults because traversing the same loop several times to generate tests produces test suites with similar test cases [30]. Fully exploring loops it is usually a goal of other testing stages such as stress testing which is out of the scope of this work. Furthermore, by considering them, the algorithm would produce a large, infinite and not practical suite.

3.3.3 Interruption Test Case Selection

Exhaustive interruption test case generation is impractical due to the large amount of generated test cases. Particularly, in the mobile phone applications context, the majority of test cases are manually executed. In this scenario, test case selection strategies are much needed. The strategy used to reduce the test suite is a test case selection based on purposes. This strategy focuses on a coverage selection criterion, the test purpose, in order to test a particular system functionality [41; 50; 51; 83; 94]. The defined test purpose is used to filter out the model, that is, it is used to remove all paths that do not lead to the desired behaviour to be tested. After that, the generation algorithm is executed, for then, generate the test cases. Formally, a test purpose can be defined as follows.

Test purposes can be specified using a simple notation, where they are defined through transition sequences. In these sequences, an "*" (asterisk) indicates that, at this point, any transition can occur. A test purpose always finishes with a transition that has an *Accept* label (indicating that all test cases need to be in conformance with the purpose) or a *Refuse* label (otherwise).

Definition 3.5 (Test Purpose). *A test purpose is a deterministic LTS* $TP = (Q^{TP}, L^{TP}, q_0^{TP}, T^{TP})$*, equipped with the special labels Accept, Refuse, and "*", and with the same alphabet as the specification, i.e.,* $L^{TP} = L^S$. Q^{TP} *is a countable, non-empty set of states,* $q_0^{TP} \in Q^{TP}$ *is the initial state, and* T^{TP} *is the transition relation.* ◇

Some hints on how to define test purposes are presented below:

- Choose the behaviour to be observed in the implementation and identify its description in the specification;

- If the behaviour to be observed is the first behaviour of the specification, then the test purpose should start with the description of this behaviour. Otherwise, add an asterisk followed by the description of the behaviour to be observed. This indicates that any behaviour can occur before the observation of the desired behaviour;

- If there are more behaviours to be observed in the same test purpose, go back to the first step;

- If the last behaviour description added to the test purpose is the last behaviour of the specification, then go to the next step. Otherwise, an asterisk should be added to the test purpose. This indicates that any other behaviour can occur after the desired behaviour;

- The last step is to add an *Accept* or a *Reject* label to the test purpose. As mentioned before, the *Accept* label is used to indicate that all generated test cases must be in conformance with the test purpose. The *Reject* label is used otherwise.

As an example of a test purpose, we will use that ALTS model from Figure 3.7 in order to define a test purpose for a scenario where a message is not removed because it is blocked. For this scenario, the following purpose could be defined: "*;'Blocked messages cannot be removed' dialog is displayed;*;Accept". The LTS model that represents this test purpose is showed in Figure 3.9.

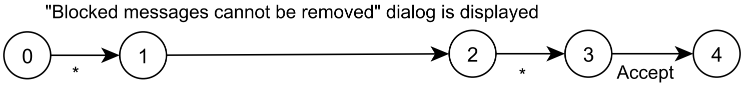


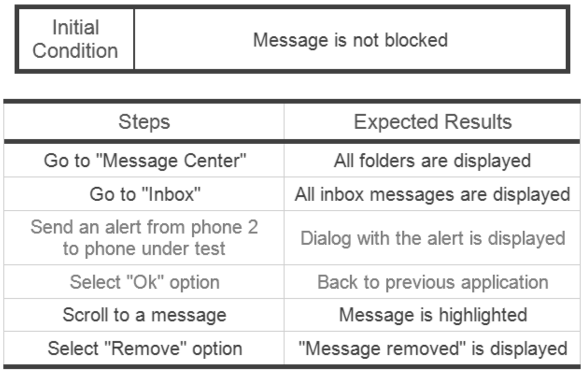
Figure 3.9: LTS Model of a Test Purpose

It is very simple to define test purposes where an interruption can occur. Given that the behaviour to be interrupted has been chosen, the name of the interruption must appear immediately after the description of this behaviour in the test purpose. The ALTS model with interruptions from Figure 3.8 will be used to demonstrate how to define test purposes to check specific interruptions. A test purpose will be defined to test the scenario where an alert appears when the user is accessing the inbox folder. This scenario can be specified through the following test purpose: "*;All inbox messages are displayed;Incoming Alert;*;Accept".

Considering the defined test purpose, the model from Figure 3.8 is filtered out to be in accordance with it. So, the following edges of the model are removed: *beginInterruption_0*, *endInterruption_0*, *beginInterruption_2*, *endInterruption_2*, *beginInterruption_3*, *endInterruption_3*, *beginInterruption_4*, *endInterruption_4*, *beginInterruption_5*, and *endInterruption_5*. The last step is to execute the test case generation algorithm.

All presented algorithms are implemented in the LTS-BT tool [29]. In order to make the test execution activity easier, considering that this activity is manual, the tool generates test cases in an alternative representation instead of ALTS. Each selected test case is transformed in a matrix, where each condition is considered as an initial condition to execute the test case.

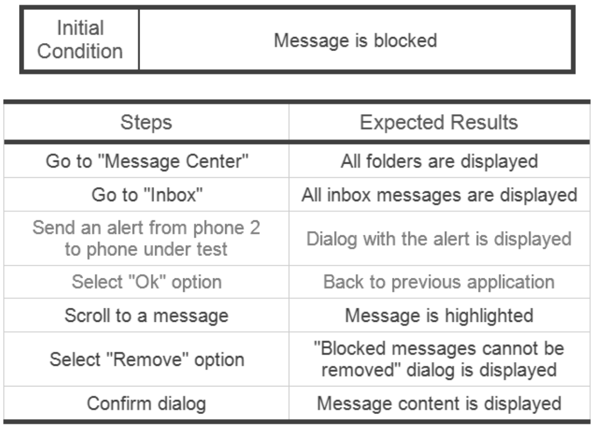
Figures 3.10 and 3.11 present the generated interruption test cases for the example above (the scenario where an alert appears when the user is accessing the inbox folder). Note that, in both generated test cases, the interruption occurs when the user is viewing the inbox folder, as was specified by the test purpose. Moreover, all scenarios of the main feature are covered. In the test case of Figure 3.10, an interruption occurs in the scenario where the message is removed, whereas, in the test case of Figure 3.11, an interruption occurs in the scenario where the message is not removed because it is blocked.



Initial Condition	Message is not blocked

Steps	Expected Results
Go to "Message Center"	All folders are displayed
Go to "Inbox"	All inbox messages are displayed
Send an alert from phone 2 to phone under test	Dialog with the alert is displayed
Select "Ok" option	Back to previous application
Scroll to a message	Message is highlighted
Select "Remove" option	"Message removed" is displayed

Figure 3.10: Test Case 01



Initial Condition	Message is blocked

Steps	Expected Results
Go to "Message Center"	All folders are displayed
Go to "Inbox"	All inbox messages are displayed
Send an alert from phone 2 to phone under test	Dialog with the alert is displayed
Select "Ok" option	Back to previous application
Scroll to a message	Message is highlighted
Select "Remove" option	"Blocked messages cannot be removed" dialog is displayed
Confirm dialog	Message content is displayed

Figure 3.11: Test Case 02

Notably LTS-BT allows for a systematic and less error-prone coverage of all possible interruptions automatically, since the tester does not need to specify all possible points where an interruption can occur – this is assumed by the tool. Moreover, LTS-BT makes it easier to focus on particular points to be interrupted and interruptions. These LTS-BT characteristics allow the tester to obtain test cases in a faster and reliable way.

It is important to remark that in the current version of the LTS-BT tool, test cases are selected as paths in the ALTS model. Therefore, even though the model is capable of representing non-determinism, the tool suits only deterministic applications, differently from the TGV tool [62], where a test case is a graph that can represent non-determinism. This is currently being addressed for the next versions of the tool.

As an example of a test case execution, the steps required for executing the test case of Figure 3.11 are: (1) the first step is to satisfy the initial condition, then an inbox message must be blocked for the test case execution. Moreover, as the test case needs an incoming alert interruption, two mobile phones must be available for the test case execution; (2) with the phone under test, the TESTER begins to execute the actions described in the test case. Thus, the “Message Center" application is started and, as result, all folders are displayed; (3) next, the second action is executed: the “Inbox" folder is selected and all inbox messages are displayed; (4) at this moment, the TESTER must cause an incoming alert interruption. So, he takes another phone and sends an alert to the phone under test. As expected result, a dialog must appear at the phone under test; (5) the TESTER selects the “Ok" option at the phone under test and control goes back to the previous application. In this case, all inbox messages must be displayed again; (6) the TESTER scrolls to a message and it is highlighted; (7) the “Remove" option is selected by the TESTER and the following message is expected: “Blocked messages cannot be removed"; (8) the TESTER confirms the dialog and the content of the message is displayed. Finally, if for all steps of the test case, the expected results were observed, the execution finishes with a pass verdict.

3.4 Properties of the Interruption Test Cases

This section comments on properties of the interruption test cases generated by the test case generation algorithm presented in the last section. Considering the execution of a test case against a SUT, three kinds of verdicts can be obtained indicating that the SUT should be approved or not: if the SUT emits the specified outputs for each input emitted by the test case, the verdict is *Pass*; if at least one of the outputs of the SUT is not specified by the specification, the verdict is *Fail*; and the *Inconclusive* verdict is emitted when the observed behaviour of SUT conforms to the specification but the behaviour described by the test purpose is not exhibited by the SUT.

It is very important to formalise the execution of the test cases in order to establish some properties as soundness and exhaustiveness, where the conformance relation is linked to verdicts obtained during the test execution [62]. Interruptions are clearly asynchronous events, but as we are considering a test architecture where the environment is fully controllable by the tester, all interruptions can be analysed as synchronous events. Thus, test cases interact with the SUT through a synchronous communication, where the execution of a test case against a SUT is modelled by a parallel composition with synchronisation on common actions. Basically, parallel composition is defined by the following rule:

$$P \mid\mid Q = \frac{p \xrightarrow{a}_P p', \; q \xrightarrow{a}_Q q'}{(p,q) \xrightarrow{a}_{P\mid\mid Q} (p',q')}.$$

Considering the defined model of test case execution, each trace $\sigma \in$ Traces$(TC$ $\mid\mid$ SUT$)$ is associated with one of the following scenarios:

- If, at any moment, any unspecified output is emitted by the SUT, the execution is stopped and the resulting verdict is *Fail*, that is, verdict$(\sigma) =$ *Fail*;
- If the SUT, at any moment, blocks or spends a lot of time to emit an output, the resulting verdict is *Inconclusive* (a timer must be used in this case). So verdict$(\sigma) =$ *Inconclusive*;
- If the outputs of the SUT are specified by the specification but the behaviour specified by a test purpose is not exhibited, the resulting verdict is *Inconclusive*, that is, verdict$(\sigma) =$ *Inconclusive*;
- If all steps of the test case are executed and all expected results are observed, then the resulting verdict is *Pass*, i.e. verdict$(\sigma) =$ *Pass*.

Given the possible situations with their respective verdicts, the rejection of a SUT by a test case TC is defined as follows.

Definition 3.6 (may reject). *TC may reject SUT $\triangleq \exists\sigma \in$ Traces$(TC \mid\mid$ SUT$)$: verdict$(\sigma) =$* Fail. $\diamond$

The conformance relation of a SUT with respect to a specification S is decided based on verdicts obtained with the execution of the generated test cases. So, the next definition formally relates the previously defined conformance relation (Definition 3.3) to the verdicts of these executions considering some properties of test cases and test suites.

Definition 3.7 (Soundness and Exhaustiveness). *A test case TC is sound for S and **conf** if $\forall$SUT, SUT **conf** $S \Rightarrow \neg(TC$ may reject SUT$)$. A test suite is sound if all its test cases are* *sound and it is exhaustive for S and* ***conf*** *if* $\forall$*SUT,* $\neg$*(SUT* ***conf*** S*)* $\Rightarrow \exists TC : TC$ *may reject SUT. Finally, a test suite is complete if it is both sound and exhaustive.* $\diamond$

Informally, a test suite is said to be sound if all correct implementations, and possibly some incorrect implementations, pass in the test (a sound test suite never rejects a correct implementation). On the other hand, a test suite is said to be exhaustive if all non-conforming implementations, and possibly some correct implementations, will not pass in the test. A test suite that can identify all conforming and non-conforming implementations is called complete.

A complete test suite is a very strong requirement for practical testing. Then, weaker requirements are accepted. In practice, sound test suites are more commonly accepted, since rejection of conforming implementations, by exhaustive test suites, may lead to unnecessary debugging. In this context, the test cases generated by LTS-BT have some properties stated in Theorem 3.1.

Theorem 3.1. *For every specification S, all test suites generated by the approach proposed in this chapter are sound. Moreover, the test suites can be considered as being exhaustive when they are generated using test purposes.* $\diamond$

The proof of Theorem 3.1 is not detailed here but the main ideas are discussed (see detail proofs in Appendix A). For soundness, we need to prove that if a test case TC may reject a SUT (implementing the specification S), then $\neg$(SUT **conf** S). In this case, we only need to prove that a *Fail* verdict of a test case only occurs if the SUT emits an unspecified output. This was already discussed in this section and the unique case where a *Fail* verdict is obtained during a test case execution is exactly when the SUT emits an unspecified output. For exhaustiveness, we need to prove that for every non-conforming SUT there is a test purpose TP and a way of generating a test case TC from S and TP, such that TC may reject SUT. Given that $\neg$(SUT **conf** S), then there is a trace σ of S such that an output of SUT after σ is not allowed by S. So, the trace σ can be used to define a TP, after that, this test purpose can be used to generate test cases where the SUT may be rejected.

3.5 Case Study

The objective of this section is to present a case study performed in order to evaluate a practical application of the approach proposed in this work. As previously said, a scenario where interruptions are allowed may have infinite test cases. Thus, in practice, only a subset of interruption test cases are manually generated and executed. Considering this context, the main goal is to compare the manual process of test case generation with the automatic process based on the algorithms presented in Section 3.3 and implemented by LTS-BT [29]. Moreover, as the amount of test cases is large, some test case selection strategy is needed. Particularly, in this case study, the strategy used to select the test suite is based on test purposes defined in order to cover a fault model which describes the set of known defects found in the past [18]. Testers use fault models to define effective test cases since the test cases are specially defined to uncover defects that are likely to be present. The use of fault models for comparing testing approaches is important in our context because only stable versions of the software are available, that is, versions with all known defects already removed. In this case, fault models allow to compare testing approaches by observing if defined test cases would uncover defects or not.

This case study was performed using applications of the mobile phone domain whose descriptions are presented in the next subsection (Subsection 3.5.1). Subsection 3.5.2 describes how the case study was defined and conducted. Finally, Subsection 3.5.3 presents the results obtained during the case study execution.

3.5.1 Overview of the Case Study Applications

This subsection briefly describes the features used during the case study. All of them are reactive applications of the mobile phone domain. In summary, the description of the features is:

Aircraft Mode This is the main feature of the case study and it is the feature that must be interrupted. Aircraft Mode feature provides the functionality of allowing the user to turn off the radio frequency transceiver and still be able to use the applications of the phone. This feature allows the user to use applications of the phone while flying in an aircraft, but without receiving calls, messages, and so on.

Table 3.1: Features

Features	Number of Use Cases	Number of Scenarios
Aircraft Mode	7	22
Incoming Call	1	2
Incoming Message	1	2
Alarm Clock	1	3

Incoming Call This is an interruption feature. Incoming Call feature provides the functionality of receiving calls.

Incoming Message This is an interruption feature. Incoming Message feature provides the functionality of receiving messages.

Alarm Clock This is another interruption feature. Alarm Clock provides the functionality of generating alarm notifications based on specific time chosen by the user.

Table 3.1 shows some metrics of the features in order to illustrate their complexity such as the number of use cases and the number of possible scenarios. It is important to remark that, considering the relationship between all features of this simple case study, the amount of interruption test cases is more than forty million tests.

3.5.2 Case Study Definition

In this subsection, we present the evaluation criteria and fault model defined for conducting and evaluating the case study. The focus is on interruption testing and generation of test suites for manual execution.

The main goal is to show evidence on the benefits of automation in the interruption test case generation and selection process using the implemented algorithms. The strategy adopted to achieve the goal is to compare the manual process of test case generation with the automatic process proposed. In practice, this kind of testing is often conducted by manual processes of selection guided by expertise. Also, there are no related proposals of more systematic strategies that could make a good basis for comparison.

As the amount of interruption test cases is very large, generation and selection is guided by a fault model specification that indicates the kind of faults that can be usually be found in this kind of applications. Thus, test cases must be selected, both in manual and in automatic, with the objective to cover the whole fault model specification. The main metrics to be observed are: (1) the time spent during the test case generation and selection and (2) the coverage of an instance of the fault model specification that contains actual faults detected in products composed of these features.

The case study was conducted by three testers: one based on automatic process (named Tester 1) and two based on manual process (named Tester 2 and Tester 3). Considering the knowledge of the testers, they had good test skills and none of them knew the features under test before the case study execution. Thus, all testers performed the case study based on two kind of information:

- The specification of the features under test;
- The fault model profile.

The specification of the features under test is a document describing all use cases (Table 3.1) according to that notation presented in Section 3.1. It is important to mention that this document was prepared in five hours. On the other hand, the fault model profile was defined based on common problems related to feature interruptions and actual defects related to the features under test. The fault model given to the testers is specified in natural language and its description is defined as follows:

- After an interruption, the interrupted application does not maintain data entered by the user;
- After an interruption, the interrupted application does not continue its execution of the same point where it was interrupted;
- Possible conflicts related to the use of shared resources (screen, network, and so on);
- Problems related to interruptions immediately before enabling the aircraft mode;
- Problems related to interruptions immediately after disabling aircraft mode;
- Problems related to interruptions when the aircraft mode is enabled.

Table 3.2: Metrics

Metrics	Tester 1	Tester 2	Tester 3
Preparation time	2 h	2 h	2 h
Generation time	80 *min*	165 *min*	150 *min*
Number of TCs	115	15	12
Productivity	86,5 TCs/h	5,4 TCs/h	4,8 TCs/h
Fault model coverage	57,14%	28,57%	28,57%
Number of invalid TCs	0 (0%)	4 (26,66%)	4 (33,33%)
Number of ineffective TCs	23 (20%)	13 (86,66%)	7 (58,33%)
Smallest TC (number of steps)	3	5	6
Biggest TC (number of steps)	12	11	8
Most common TC size	5	6 and 8	8

By measuring coverage of an instance of this specification with actual faults (instead of coverage of the specification), where one kind of fault may correspond to more than one actual fault, it is possible to analyse which approach can be more effective to systematically investigate the implementation by generating a more complete test suite.

3.5.3 Case Study Results

This subsection presents and discusses the obtained results. Table 3.2 presents the metrics collected during the case study execution.

The first step of the case study consisted in reading the specification and the fault model. These documents are usually constructed prior to the testing process by requirements and quality engineers. In this sense, all testers had the same preparation time (Table 3.2, line "Preparation time"), and as previously said, Tester 1 generated the test suite through an automatic process by using LTS-BT [29], and Tester 2 and Tester 3 generated the tests through a manual process. According to the results, from Table 3.2, Tester 1 generated the tests in less time (line "Generation time"). The generation time of Tester 1 basically consisted of the time needed to define test purposes once each LTS-BT execution consumed less than one second to generate test cases. Considering the generation time of Tester 2 and Tester 3, they spent more time because all test cases were manually selected and written in the format shown in Figures 3.10 and 3.11. Furthermore, Tester 1 generated the largest test suite (line “Number of TCs”, where TCs means Test Cases) implying in more productivity (line “Productivity”). The productivity was calculated by observing the number of test cases generated per hour.

Considering the fault model coverage (Table 3.2, line “Fault model coverage”), Tester 1 reached the best coverage. It is important to remark that the percentage of fault model coverage achieved by Tester 2 and Tester 3 are equal but the faults found by them are not the same. Nevertheless, the set of faults found by Tester 1 contains all faults found by the other testers. This result is expected as LTS-BT allows a more systematic test case generation process, from the same base specification. However, as the process is guided by test purposes defined by the tester, the fault model coverage depends also on the tester’s experience.

On the other hand, test cases generated through manual process are error-prone. The line “Number of invalid TCs” of Table 3.2 shows the number of test cases generated with errors, that is, test cases impossible to run, mainly because they miss information. Moreover, manually generated test suites tend not to take all scenarios of an interruption into account. This does not occur in the automatic process because when the tester decides, for example, to check the incoming call interruption at some point of the feature under test, the developed algorithms consider all scenarios of the interruption, for example, when the call is accepted and when it is rejected by the user.

Considering the number of test cases that actually do not find defects (line “Number of ineffective TCs”), Tester 1 reached the best results. The number of ineffective test cases also considers the invalid test cases. Finally, the three last lines of Table 3.2 give information about the size of generated test cases. Note that test cases generated by both strategies are similar w.r.t. size, that is, number of steps. This is explained by the fact that test cases were generated based on a structured document that may induce the same general kind of test cases to be defined. In practice, manual testing is not usually based on structured documents and then test cases tend to be as simple as possible. However, not using the same input document would put a threat to validity of the results, in the sense that with the same inputs, both strategies had the same basic information available.

In the scope of this case study, it is possible to conclude that the proposed strategy and tool allow a more systematic test case generation process contributing to better productivity and effectiveness of test process, depending on the tester's experience. Moreover, some problems of the manual process such as erroneous test cases generation is solved by the automatic process since all generated tests are sound (Section 3.4). However, note that Tester 1 did not reach 100% coverage. For this, it is necessary to define complete test purposes regarding the fault model. As this depends on the tester's experience and accuracy, it is possible that he can miss behaviour that should have been considered.

It is important to remark that the problem of selection in the scope of integration testing in general is a hard one. The possible number of test cases resulting from different combinations is large and also the set of all combinations is usually intractable by manual investigation. As a consequence, the achieved level of fault coverage depends greatly on information and expertise available to pinpoint the key test cases, for instance, common faults detected in a domain. For instance, only 57% of faults were covered by the test suite generated. However, note that this further exceeds the manually generated suites. Even though the testing strategy presented in this chapter is based on automatic generation, it also allows the experienced tester to target the selection process by defining the test purposes in a systematic way.

3.6 Related Work

This section presents some works related to our proposal. Lorentsen et al. [88] propose a way of identifying categories of interactions and create behavioural models that capture those interactions, where interruptions are a type of interaction. They use Coloured Petri Nets to manually model the interactions and a model checker for interactive graphical simulation. As disadvantages, the process is manual and the work is not devoted to testing.

Another interesting work is that belonging to Jard and Jéron [62], where the TGV tool is presented. TGV receives a specification and a test purpose as input and produces abstract test cases as output. The TGV input format for both specification and test purpose is IOLTS (already defined in Subsection 3.2.1). As mentioned in Subsection 3.2.1, it is possible to represent interruptions through IOLTS models. So the TGV tool can be used to generate interruption test cases, but an interruption behaviour needs to be replicated if it can occur at more than one place. Moreover, it is not possible to directly represent conditions associated to actions and due to the fact that the same interruption behaviour is replicated in the IOLTS model, the test purpose must specify the point where we want to verify the interruption and all other points where the interruption cannot occur. Thus, given that the tester needs to manipulate LTS models in the definition of the TGV test purposes, this notation is not useful in practice for interruption testing.

One possible solution is to consider the tool set proposed by the AGEDIS project that can generate test cases from high level models (e.g. UML diagrams) [57], where TGV is internally used to generate test cases. However, the tool set does not support interruption specifications directly as well as the newest version of UML (UML 2.0) with its greatly improved diagrams. In this case, the difficulties with interruption modelling and test purposes definition remain.

The process algebra CSP (Communicating Sequential Processes) was designed for describing systems of interacting components [109]. CSP has a specific operator for describing interruptions but its semantics is very different from the high (application) level interruption notion addressed in this chapter. The CSP interruption operator specifies that when a process P_1 is interrupted by another process P_2, the process P_1 is discarded and P_2 begins its execution. In our context, the process P_1 executes again after the execution of the process P_2. Jovanovic et al. [68] have proposed an extension of CSP to represent this kind of behaviour but there is not any tool supporting their proposal. Figueiredo et al. [39] present a behavioural model that represents interruptions in CSP without using the interruption operator, but the presented model is more suitable for representing the semantics of interruption test behaviour as presented in this chapter. Nogueira et al. [102] propose an approach to test case generation based on CSP. The SUT is specified using the same use case template presented in Section 3.1 and it is automatically translated to CSP using the strategy presented in [26]. The main objective of the work presented in [102] is to provide a strategy for testing individual features and feature interactions. Moreover, test purposes are defined in low level using CSP. The interruption testing is not directly treated, but the work can be adapted to test interruptions considering that all the possible points of interactions are specified in advance. In our approach, all points of interruption do not need to be explicitly specified, making the work of the tester easier and less error-prone.

Furthermore, the work presented in [25; 34] propose a strategy to reduce the test suite size based on test cases prioritisation. This requires that every individual pair of interactions is included at least once in a test suite. In this case, if it is not possible to execute the entire test suite, the tester can execute at least the most important test cases. But note that the prioritisation information is given by the tester. The strategy presented in this chapter is similar, in the sense that the focus is on particular interruptions by using a test purpose, but due to the expressiveness of test purposes, it is also possible to focus on and/or exclude particular functionalities associated with the interruptions.

3.7 Concluding Remarks

This chapter presented an approach to interruption testing that is based on a model capable of representing interruptions for reactive systems. The model makes it possible for interruptions to be combined at different points of possibly different flows of execution. This model is supported by the LTS-BT tool along with a test case generation algorithm and a test purpose-based selection technique. Test selection is crucial for interruption testing since the number of possible test cases is enormous. Also, in practice, not all possible points of interruption are fault-prone.

It is important to mention that the current version of LTS-BT is restricted to deterministic systems. This may seem unrealistic. However, particularly, if embedded systems such as mobile phone applications are considered, the tool can be largely applied. For these systems, applications are often deterministic ones that run on single-processor, single and restricted screen, and so on. However, they have complex patterns of interruptions which clearly justify the need for modelling and systematic test selection. Furthermore, ALTS models are capable of representing non-determinism and the algorithms can be clearly extended to support non-deterministic systems since the semantics of ALTS and IOLTS are very similar.

Chapter 4

Related Work and Problem Statements

This chapter presents a review of relevant work on the testing of real-time systems which is concerned with deriving real-time test cases from specifications where variables and actions with parameters are allowed. As research related to the test of real-time systems is very recent, there are few works aligned with this challenge. After the analysis of the related work, several limitations and open problems are identified.

4.1 Related Work

Since strategies to generate real-time symbolic test cases are practically nonexistent, the focus here is to describe approaches related to testing real-time systems (not exactly symbolic testing of real-time systems) and argue why they are not considered as symbolic testing strategies.

4.1.1 Cardell-Oliver

Cardell-Oliver's work [28] addresses the problem of conformance testing for real-time systems and proposes an approach, based on UPPAAL timed automata specifications [82], to testing the same kind of system. As the UPPAAL timed automata have a dense or analogue-time model, Cardell-Oliver argues that their traces include behaviour which cannot be observed in an experiment. An example of an analogue-clock test case is: provide an input at time 1 and expect for the result at time 3. The tester implementing this test must be able to emit the input precisely at time 1 and check whether the output occurred precisely at time 3. In practice, the tester has finite-precision clocks and sample the outputs of the system under test periodically, e.g. every 0.3 time units, thus, it cannot distinguish between the output arriving anywhere in the interval (2.9, 3.1). In this sense, it is very difficult, if not impossible, to implement analogue-clock tests using finite-precision clocks. In this case, Cardell-Oliver proposes that a more appropriate model for observing real-time systems is a digital clock approximation.

As a TLTS representation of a TA can possibly have infinite states because of the representation of time, each timed trace of the TLTS is mapped into a set of possible integer-timed trace interpretations. Thus, symbolic states are used to represent a set of clock valuations. The symbolism is only used to abstract time and does not take the data of the system under test into account, so this proposal cannot be classified as a symbolic testing approach.

Furthermore, the paper considers a kind of test purpose, named test views, where the tester can select relevant events to observe. An implementation relation is defined based on trace equivalence under the assumption that the implementation is input-enabled (input-complete) and that it has no more states than the specification. An algorithm implementing the approach is presented, but, it seems that there is not a tool to support the work.

4.1.2 En-Nouaary et al.

En-Nouaary et al. [46] address the issue of testing real-time systems specified as a variant of the TAIO presented in Subsection 2.2.3. The TAIO considered in [46] is assumed to have instantaneous transitions, that is, once the transitions are enabled they must be taken immediately. This assumption reduces too much the expressiveness of the model, for example a simple specification such as “when an input is provided, an output must be generated within at most 10 time units” cannot be expressed.

En-Nouaary et al. propose a test case generation approach that is divided into three steps: firstly, as the semantics of a TAIO can be defined in terms of an infinite TIOLTS (see Subsection 2.2.3), the authors represent this infinite TIOLTS using a finite region graph where the locations symbolically represent a set of clock valuations. Secondly, each clock region of the region graph is sampled, according to a granularity, in a way that each clock region is transformed into a finite set of clock valuations; thus, the region graph is reduced to another graph, named grid automaton, which is then transformed into a timed finite state machine. Finally, they use state characterization techniques for test case generation [47].

Note that, as the work only represents time symbolically, it is not considered as a symbolic testing strategy, since the data of the system must be also symbolically taken into account. The work supposes that the implementation under test has the same number of locations as the specification. Thus, the implementation relation is based on trace equivalence. The adopted assumption is very strong compromising the usefulness of the work in practice. Moreover, no algorithms for test case generation are shown.

An interesting contribution of En-Nouaary et al. is the study about fault models. They argue that two types of faults are possible: timing faults and action and transfer faults [48]. Timing faults are related to violation of transition time constraints and action and transfer faults are similar to the classical faults of finite state machines. In [46], the authors discuss how the fault model can be used to test timed systems based on TAIO model.

In [44; 45], the testing approach is improved changing the last step of the test case generation process. Instead of transforming the generated grid automata into a timed finite state machine, the grid automata is traversed using an adaptation of the Depth-First Search strategy in order to generate test cases. In this more recent work, a test selection strategy based on test purposes is defined and algorithms are presented. Nevertheless, this improved approach only represents time symbolically.

4.1.3 Li et al.

Li et al. [87] propose an approach to property-oriented real-time test case generation. As specification language, they use time-enriched statecharts and provide a restricted real-time logic as the property specification language. Li et al. argue that statecharts cannot be easily manipulated for test generation, then they provide a way of transforming statecharts into extended finite state machines, from which test sequences are obtained.

Li et al. [87] focus only on specification languages. Thus, several concepts of a complete testing strategy are not taken into account such as assumptions related to specifications and implementations, conformance relation, test architecture, tools, case studies, and so on.

4.1.4 Khoumsi

Khoumsi [69; 70] proposes an approach to symbolic test case generation for real-time systems. To the best of our knowledge, this is one of the few approaches that tries to symbolically deal with time and variables and actions with parameters representing the system data. The Khoumsi's main objective is to combine a real-time testing strategy with a non-real-time symbolic testing strategy.

Khoumsi's work is based on that symbolic model theory presented in Subsection 2.1.9 extended with time. Basically, the proposed approach is divided into two steps: firstly, the real-time symbolic model is transformed into an automaton where the setting and expiration of clocks are represented as actions; finally, the symbolic testing approach presented in Subsection 2.1.9 is adapted to generate test cases.

In this approach, a new real-time symbolic model is proposed, named Timed Input-Output Symbolic Automata (TIOSA), but its semantics is not formally defined. Additionally, the first step of the approach, described above, restricts too much the use of clocks, guards, and clock resets leading to a less expressive and flexible specification language. Under the assumption of input-completeness, the adopted conformance relation can be considered as timed trace inclusion. Finally, there is no tool supporting the work and real case studies were not performed to validate the applicability of the proposed strategy.

4.1.5 Briones and Brinksma

Briones and Brinksma [23] present en extension of Tretmans' theory and algorithm [117] for testing real-time systems. A distinguishing characteristic of this work is that it takes quiescence into account and provides an operational interpretation of this concept in the context of RTS. Only output quiescence is considered. They consider an output quiescent state as one where the system is unable to generate an output without further input stimuli. Briones and Brinksma argue that the quiescence can only be detected by waiting for outputs, but as we cannot wait forever a maximal duration M must be defined. So, the work proposes a parameterised conformance relation where output quiescence only can be observed after a minimal delay of M time units. The work is based on TIOLTS, which serves as semantics for TAIO (see Subsection 2.2.3). Additionally, the paper defines the concept of real-time test cases considering execution and verdicts, and presents in an abstract way an algorithm for generating them. Considering that the underlying continuous model of time is represented through an infinite TIOLTS, the proposed algorithm generates uncountable test cases.

In [24], Briones and Brinksma extend the framework proposed in [23] in the sense of allowing the implementation to be sometimes non-input-complete. This new paper presents an extension of TIOLTS where input and output sets are divided in channels and in each reachable state each input channel is either blocked or all inputs are accepted, i.e. the implementation can sometimes be non-input-enabled. The general maximal duration M, cited above, is relaxed and they allow different bounds for different sets of outputs. Moreover, the entire framework is updated to deal with these extensions, including the algorithm for test case generation. But, the algorithm has the same problem as in [23], that is, it also generates uncountable test cases.

4.1.6 Bohnenkamp and Belinfante

Bohnenkamp and Belinfante [20] present an extension of TorX [119] where timing constraints can be expressed in the specification. TorX is an on-the-fly testing tool that tests for the conformance relation proposed by Tretmans [117]. Bohnenkamp and Belinfante's work is influenced by the framework presented in Subsection 4.1.5, that is, the main objective of this work is to provide an extension of TorX to implement the main ideas of the Briones and Brinksma's work [23].

To avoid the generation of uncountable test cases, Bohnenkamp and Belinfante adopt a symbolic representation of TIOLTS, where each symbolic location represents the maximal set of clock valuations that satisfy a given clock constraint. This strategy is not considered as a symbolic testing strategy, as only time is abstracted. The tool assumes input-completeness of the IUT and that the tool must run on the same host as the IUT. The latter restriction reduces the usefulness of the tool in practice, since several systems cannot be tested such as embedded systems with limited resources (e.g. smart cards, mobile phones, music players, and so on).

4.1.7 Bodeveix et al.

Bodeveix et al. [19] propose a way of checking real-time dependability requirements by means of testing. They adopted a particular kind of timed automata, where determinism and explicit inputs and outputs are assumed. The main idea is to model dependability requirements as test purposes. The strategy is described in three steps: (1) a kind of synchronous product is performed between the specification and the requirement to be checked; (2) the states of the resulting model are symbolically represented abstracting only time; (3) a reachability algorithm is executed to generate only one test case capable of checking the dependability requirement.

The paper is very short and important concepts of a complete testing framework are not discussed such as assumptions related to specifications and implementations, conformance relation, test cases, verdicts, oracles, and so on. Additionally, algorithms are not shown and tools were not developed to support the proposal.

4.1.8 Larsen et al.

Larsen et al. [80] propose a tool and the related theory for online testing of real-time systems. The work is based on non-deterministic timed automata with inputs and outputs (TAIO) specifications. However, the adopted TAIO is a variant of that presented in Subsection 2.2.3, where both locations and transitions can have guards (clock constraints). The developed tool was firstly named T-UPPAAL [100], but today its name is UPPAAL TRON (UPPAAL for Testing Real-time systems ONline). This tool was implemented by extending the UPPAAL model-checking tool [82].

A distinct characteristic of the work is the proposal of a formal implementation relation that takes environment assumptions into account. An environment assumption is a kind of test purpose of the property oriented testing theory, presented in Subsection 2.1.6. Thus, the main goal of the proposed implementation relation is to check if an implementation is in conformance with its specification when operating under some environment assumptions. According to Larsen et al. [80], modelling the assumptions separately has several advantages: (1) considering a specific environment, the testing tool generates only realistic test cases reducing the number of test cases and improving the quality of the test suite; (2) the testing process can be guided to specific scenarios of interest; (3) when the environment model is separated from the system model it is easier to test the system under different assumptions. The proposed implementation relation is based on Tretmans and de Vries' work [118; 40] and coincides with timed trace inclusion considering the input enabledness assumption.

As previously said, the work is based on a variation of TAIO, but Larsen et al. [80] show that the semantics of a TAIO can be defined in terms of an infinite TIOLTS. Thus, all the theories and algorithms presented are based on TIOLTS. In [80], the main algorithm for generating and executing test cases are presented as well as its proof of soundness and completeness. As a TIOLTS, representing a TAIO, can possibly have infinite states because of the representation of time, the generation of test cases uses a reachability algorithm that operates on symbolic states (a symbolic state represents a set of clock valuations). As only time is symbolically represented and the system data is not symbolically taken into account, we cannot consider this proposal as a symbolic testing approach.

The work reported in [80] presents an experiment to validate UPPAAL TRON and indicate the applicability of the proposed technique. As the test generation and execution is online, a drawback is the need to implement an adapter component to link the system under test to the UPPAAL TRON tool. In order to evaluate the algorithms and the tool in detail, a real-life application was tested and the results are described in [81].

4.1.9 Hessel et al.

Hessel et al. [60] present a strategy of time-optimal test case generation using the data structures and algorithms of the UPPAAL model checking tool [82]. Time-optimal test cases are defined as test cases guaranteed to take the least possible time to execute. According to Hessel et al. [60], time-optimal test cases are important for some reasons: (1) using time-optimal test cases, the total execution time of a test suite is reduced allowing more behaviour to be tested; (2) regression testing can be executed as quickly as possible; (3) the time-optimal test cases have high probability of detecting errors, considering that the fastest scenarios are stressful situations. The proposed strategy can generate test cases using manually defined test purposes or automatically generated from some coverage criteria.

The proposed strategy assumes that both the implementation under test (IUT) and the environment (the test purpose) are modelled using a deterministic and output urgent class of TA. A TA is deterministic when two transitions with the same label lead to the same state and it is output urgent if, when an output is enabled, it occurs immediately. These characteristics are very restrictive reducing the expressiveness of the model. The work uses the fastest diagnostic trace of the UPPAAL tool to generate time-optimal sequences. This functionality of UPPAAL generates a trace with the shortest accumulated time delay witnessing a submitted safety property. Then, a test case is generated from this diagnostic trace. As previously said in Subsection 4.1.8, the UPPAAL's reachability algorithm operates on symbolic states, but only time is symbolically represented. The implementation relation considered is the timed trace inclusion as in [80; 81]. One of the main ideas of the work is that test purposes can be formulated as safety properties that can be checked by the reachability analysis performed in a model generated by the combination of the specification with the test purpose. Another interesting idea is the use of coverage criteria to generate test cases. In this case, the proposed solution is to annotate the model using auxiliary variables to mark the coverage of target elements (e.g. edges and locations). The work also presents an interesting experiment, but it is clear that the annotation of the model with new auxiliary variables is tedious and error prone in practice. Moreover, the inclusion of these variables increases the state space reducing the applicability of the work in practice.

Several limitations of the work described above are solved in another work [61], which extends the requirement specification language of UPPAAL in order to allow the use of keywords to represent coverage criteria. Therewith, the reachability analysis algorithm of the UPPAAL tool was modified to eliminate the need of manually annotating the model and another version of UPPAAL was created, the UPPAAL CoVer tool. Even so, the defined language is still very restrictive and the specification model continues being deterministic and output urgent.

Finally, a book chapter [59] was written to describe both the online testing using UPPAAL proposed by Larsen et al. [80; 81] and the offline testing using UPPAAL proposed by Hessel et al. [60; 61].

4.1.10 Merayo et al.

Merayo et al. [96] present a formal framework to specify and test real-time systems that considers both hard and soft deadlines, where hard deadlines must be always met on time and soft deadlines can be sometimes met in different times. The model used to specify the software system is an extension of the classical finite state machine, called Timed Extended Finite State Machines (TEFSM). Transitions in classical finite state machines indicate that if a machine is in a state s and receives an input i then an output o will be produced and it will change its state to s' (this can be represented as $s \overset{i/o}{\rightarrow} s'$). The timed extension proposed is represented as $s \overset{i/o}{\longrightarrow}_{[t_1,t_2]} s'$ and it means that if a machine is in a state s and receives an input i then an output o will be produced and it will change its state to s' wasting a time greater than or equal to t_1 and smaller than or equal to t_2. Merayo et al. argues that, in the context of RTS, the notion of correctness has several possible definitions. For this, the framework defines several conformance relations. In practice, one may consider that a system under test is in conformance with a specification if all actions are performed exactly on a predefined time, while another could consider that the implementation has to be always/sometimes faster. Thus, depending on the situation, a different conformance relation can be taken into account. The presented conformance relations are based on Tretmans' work [117]. The paper also formalizes the concept of test cases and test suites and gives some directions of how to apply the test cases to an implementation. But, algorithms are not presented.

The work discussed above was extended in [97], which itself continues the proposal presented in [103]. In this more recent work, two kinds of time are considered: actions with associated time and time-outs. When time-outs are allowed the state of the system can change only with the time evolution (without the occurrence of actions). Moreover, an algorithm to generate test cases is presented and some small examples are discussed. In [98], Merayo et al. extends the work [97] to deal with stochastic time systems. Finally, the work presented in [99] extends all the developed formal framework to deal with specifications where time requirements are defined using intervals.

The ideas presented by Merayo et al. [96; 97; 98; 99; 103] are very interesting. It is extremely useful to have many conformance relations, so the theory can be used in different contexts. But, in general, the work has some disadvantages: (1) there is no a tool supporting the approach; (2) real examples and case studies are not presented; (3) the model does not allow to specify actions with parameters and communicating machines are not considered, thus it is not possible to deal with asynchronous events; (4) only one clock is used to control how time evolves reducing the expressiveness of the model; (5) only the discrete-time model is considered.

4.1.11 Krichen and Tripakis

Currently, one of the most complete works on testing of real-time systems is the one developed by Krichen and Tripakis [73; 74; 75; 76; 77; 78]. In [74], they propose a framework for conformance testing of real-time systems where specifications are modelled as non-deterministic and partially-observable timed automata. Krichen and Tripakis argue that, in practice, when the model is built compositionally, component interactions are usually non-observable by the tester and this abstraction often results in non-determinism.

In comparison with other approaches, Krichen and Tripakis' work uses less restricted timed automata. For example, several restrict assumptions are considered by other approaches such as isolated and urgent outputs (Subsection 4.1.9); the use of determinizable timed automata with restricted clock resets (Subsection 4.1.4); the use of trace equivalence as conformance relation, considering that the implementation has no more states than the specification (Subsections 4.1.1 and 4.1.2); permission of using only one clock in the specification (Subsection 4.1.10); and so on. Krichen and Tripakis use a kind of TAIO where each transition is annotated with one of the following three deadlines: lazy, delayable, and eager. The lazy deadline imposes no urgency, delayable means that once enabled the transition must be taken before it becomes disabled, and eager means the transition must be taken as soon as it becomes enabled.

They propose an extension of the conformance relation from [118], named timed input-output conformance (tioco). This new conformance relation is defined by including time delays in the set of observable outputs. They do not require the specification to be input-complete, thus tioco is more expressive than the other conformance relations such as trace equivalence (Subsections 4.1.1 and 4.1.2) and trace inclusion (Subsections 4.1.9 and 4.1.4). The proposed conformance relation is more expressive in the sense that it allows an implementation to accept inputs not accepted by the specification, whereas the other timed conformance relations above do not. Several characteristics of tioco are discussed in [77] such as transitivity, an extensive comparison with related conformance relations, the prove that checking tioco is undecidable (it is not a problem for black-box testing since the implementation model is unknown, the conformance cannot be directly checked) and that it does not distinguish specifications with the same set of observable traces.

Most of the other authors only consider analogue-clock tests, i.e. tests that are very difficult to execute in practice. Nevertheless, Krichen and Tripakis consider both analogue-clock and digital-clock tests. Analogue-clock tests can measure precisely the delay between two events, whereas digital-clock tests can only count how many time units of a periodic clock have occurred between two events [74]. They use symbolic reachability algorithms for test case generation, whereas other approaches use symbolic representation of time with classical reachability algorithms. There is a tool, named TTG, supporting the proposed strategy and in [74] only a toy example is used as case study.

The strategy of analogue and digital-clock test generation is improved in [75]. Most test generation algorithms rely on an implicit determinization of the specification during the test case generation, but this is a problem when analogue-clock tests are considered because timed automata are not determinizable in general [5]. In [74], Krichen and Tripakis proposed an on-the-fly determinization of the specification during the execution of the test, but the generated algorithm is costly and the tester must quickly respond to the outputs of the system under test. Thus, they proposed, in [75], a pragmatic approach where they suppose that the tester has a single clock and that it is reset every time the tester observes an action of the system. This proposed strategy allows analogue-clock tests to be represented as deterministic timed automata. The generation of digital-clock tests has a different problem: as this kind of test can be represented statically as finite trees, the generation can be offline or on-the-fly and the strategy adopted in [74] is to generate all possible tests up to a given depth, leading to an explosion of test cases. In [75], this problem is solved by providing a method to generate tests which cover the specification with respect to some criteria.

Krichen and Tripakis present, in [76], the proposed framework in a methodological point of view with emphasis on the expressiveness of the models and showing several examples of scenarios to be specified. A real example is used as case study in this more recent work. Finally, all the work discussed in this section is described in detail in Krichen's PhD thesis [73] and in [78].

4.1.12 Zheng et al.

Zheng et al. [124] address methods to generate test cases from formal specifications of real-time systems and provide a metric-based test selection method for sufficient testing of a given implementation. The systems are specified using an object-oriented approach: firstly, the abstract data types are separately defined; secondly, each reactive object of the system is specified using a kind of extended finite state machines; finally, the whole system is specified as a network of communicating objects.

As the specification notation allows time to be continuous, a grid is used to digitise the extended finite state machines. The grid is a covering of the underlying analogue time space, mapping points of that space onto a single representative of each grid region. So, this strategy also represents time symbolically as most of the other approaches discussed in this chapter. From grid automata, test cases can be generated according to a given coverage criterion (state or transition coverage) or according to a fault model.

The work also describes an experimental study where a test bed implementing the proposed strategy, named TROMLAB, is used to validate the work. But, the work has some drawbacks such as a conformance relation is not defined, algorithms and examples of test cases are not presented, the cited tool is not available, and the specification language only considers synchronous communication between objects.

4.1.13 David et al.

David et al. [37] propose a game-theoretic approach to the testing of real-time systems. Systems are modelled by Timed Input-Output Game Automata (TIOGA), which is a variant of timed automata (see Subsection 2.2.3) with their actions partitioned into controllable ones and uncontrollable ones. When an action is controllable it means that the tester determines when or which action will occur, whereas when an action is uncontrollable it means that it is the system under test that determines when or which action will occur. Considering that the set of actions are divided into input and output actions, David et al. assume all output actions to be uncontrollable and all input actions to be controllable. Test purposes can be defined as Timed CTL formulas.

According to David et al., the proposed strategy can be defined as follows: a play of the timed game between the system and the tester is a run of the TIOGA towards a specified test purpose; if the Timed CTL formula is satisfied by the TIOGA, it can synthesize a winning strategy; since a winning strategy is a guide towards the goal states, which satisfy the test purpose, it can be viewed as a test case. This strategy is implemented in a timed game solver, named UPPAAL TIGA.

David et al. show that the semantics of TIOGA can be defined in terms of TIOLTS and reuse the conformance relation defined by Krichen and Tripakis (Subsection 4.1.11). Internally, the strategy represents time in a symbolic way and uses model checking techniques to verify the satisfiability of a formula against a specification, and if so, a test case is generated. In this sense, this technique is not considered to be a symbolic testing strategy.

In [38], David et al. propose another game-theoretic approach to testing partially observable real-time systems. This work differs from the former because the tester may observe neither internal actions nor internal state changes due to these internal actions. Moreover, the tester has limited precision ways to analyse the SUT, which avoids knowing which state the SUT is in or the precise observation a timed trace. In this case, the SUT can only be observed through a finite number of possible observations. Finally, an observation-based conformance relation is proposed along with algorithms for test case generation, but only time is symbolically treated.

4.1.14 Adjir et al.

Adjir et al. [3; 4] propose a technique for conformance testing of real-time systems using TINA, a toolbox for the edition and analysis of Petri Nets and Time Petri Nets. According to the authors, the toolbox allows the generation of time-optimal test cases. As specification language, they use Prioritised Time Petri Nets. In this model, there is a priority relation on the transitions, that is, a transition can only be fired if it has the highest priority at the moment.

The authors mention that the proposed technique is based on timed trace inclusion, but no conformance relation is formally defined. Specific scenarios of testing can be selected through manually defined test purposes and covering criteria specified in the state-event linear temporal logic SE-LTL. Besides not presenting algorithms or case studies, this work only abstracts time.

4.1.15 Styp et al.

Styp et al. [120] propose a combination of symbolic transition systems introduced in [52] with timed automata presented in [5], named Symbolic Timed Automata (STA). The proposed formalism allows the modelling of real-time reactive systems with data input and output. It is possible to use variables as bounds in clock guards and associate clock invariants with locations. The semantics of STA is defined in terms of TLTS (see Subsection 2.2.3).

A new conformance relation is defined: stioco. The stioco relation is very similar to the tioco relation defined by Krichen and Tripakis (Subsection 4.1.11), but symbolic constraints (universally quantified formulas) are considered instead of dealing with concrete outputs. Thus, an implementation conforms to a specification for stioco, if, whenever the constraints are satisfied for the implementation to produce an output or delay, then also the specification satisfies the constraints to produce the same output or delay. The authors state that stioco coincides with tioco at the semantic level.

As the work is at the beginning, it is far from a complete testing approach. Test cases are not defined along with a test architecture to execute them. Test case generation and selection strategies are not presented. Moreover, there are neither algorithms nor tools to support the work proposed.

4.1.16 Timo et al.

Timo and Rollet [113; 114] propose a conformance testing approach to data-flow real-time systems based on a variant of timed automata in which only variable changing are considered as events.

In [114] the model called Variable Driven Timed Automata (VDTA) is proposed. In the states of a VDTA either the time elapses continuously or the environment modifies the values of input variables. All transitions are urgent and the values of output variables can only be observed. Moreover, a timed variable-change conformance relation (tvco) is proposed. Basically, an implementation is in conformance with its specification for tvco if all behaviours of the implementation are allowed by its specification. In this case, the implementation must change the value of input variables in a time allowed by the specification. An online testing algorithm is proposed, but test cases are not formally defined.

The approach presented in [114] is improved in [113]. Test purposes are introduced as a test case selection strategy and the time is treated with region graphs, which may lead to the state space explosion problem. Finally, abstract interpretation and approximation techniques are proposed to generate test cases, but no algorithms are presented. It is important to mention that only deterministic models are considered, quiescence is not discussed, and there is no tool supporting the work.

4.2 Comparison of Reviewed Work

This section concludes the analysis of related work with a comparison among all studied contributions. The analysis is divided into three tables because of the limited space. In the first table (Table 4.1), the following characteristics are considered: (1) if the strategy of test case generation is online or offline; (2) if the proposal allows test purposes specification; (3) if the proposed strategy is supported by tools; (4) the specification language used to model the IUT; (5) if the work takes quiescence into account.

Considering the first characteristic, most of the approaches adopt offline test generation. Few approaches that consider an online test case generation has a tool available. As discussed before, it is easier to deal with non-determinism through an online strategy. On the other hand, it is more difficult to guide the generation with test purposes. An interesting work is the one developed by Krichen and Tripakis (Subsection 4.1.11), where the generation is offline but the strategy takes non-determinism into account. The generated test cases can be seen as trees and the action of the tester depends on the observation history. It is important to remark that non-determinism is an important characteristic of the real-time context, since most of specifications are composed of parallel components.

The second characteristic is related to specification of properties to be verified during the test. Most of the approaches allow the specification of test purposes. In some cases, the approaches marked with an asterisk "*", the strategy allows the specification of the environment that interacts with the IUT emitting inputs and receiving outputs. In this case, the specification of the environment can be considered as a test purpose. Test purposes are extremely needed in the context of real-time systems because the available algorithms usually generate a huge amount of test cases.

As the researches in the context of real-time testing are very recent, there are few tools available. Basically, we can cite TROM (Subsection 4.1.8), CoVer (Subsection 4.1.9), TTG (Subsection 4.1.11), TIGA (Subsection 4.1.13), and TINA (Subsection 4.1.14) for effective testing of real-time systems. The other tools marked with an asterisk are only cited on their respective papers, but they are not available. The three tools TROM, CoVer, and TIGA are related in the sense that they are based on the UPPAAL model checking tool. These three tools use the UPPAAL notation as the input specification.

Considering the notation used as specification language, most of the approaches use models derived from timed automata [5]. In general, timed automata cannot be determinized [5], thus most approaches impose several restrictions to the specification language. Some authors completely disallow non-determinism such as Cardell-Oliver (Subsection 4.1.1), En-Nouaary et al. (Subsection 4.1.2), Li et al. (Subsection 4.1.3), Briones and Brinksma (Subsection 4.1.5), Bodeveix et al. (Subsection 4.1.7), Hessel et al. (Subsection 4.1.9), Zheng et al. (Subsection 4.1.12), and David et al. (Subsection 4.1.13); whereas, others restrict the use of clocks, guards, or clock resets such as Khoumsi (Subsection 4.1.4), Bohnenkamp and Belinfante (Subsection 4.1.6), and Merayo et al. (Subsection 4.1.10). The most expressive specification languages are used by Larsen et al. (Subsection 4.1.8) and Krichen and Tripakis (Subsection 4.1.11). Non-determinism is also important to model timing uncertainty, that is, it is more realistic to allow an output occurring in some interval of time. In this case, non-determinism is a choice between letting the time pass or emitting an output.

Khoumsi (Subsection 4.1.4) is one of the authors that proposes a specification language where parameters and variables containing data of the system can be defined. This is the first step in order to provide an effective real-time symbolic testing strategy. Nevertheless, strong restrictions are made on clocks, guards, and clock resets leading to restrictions on its applicability in practice. Two other approaches described in Subsections 4.1.15 and 4.1.16 are intended to provide a real-time symbolic testing strategy, but these approaches can be considered as incomplete since test cases are not formally defined, a test architecture is not defined, no algorithms are presented, and there is no tool supporting the work.

The last characteristic considered in Table 4.1 is quiescence. Quiescence is a characteristic of systems that indicates the absence of outputs, and as described in Subsection 2.1.5 it is extremely related to real-time systems. To provide an effective way of dealing with quiescence, the following concepts must take it into account: input specification, conformance relation, oracle, and so on. In this sense, only two approaches consider quiescence: Briones and Brinksma (Subsection 4.1.5) and Bohnenkamp and Belinfante (Subsection 4.1.6). However, the former work does not have an implemented tool, whereas the latter implemented a prototype which is unavailable.

Work	**Test Case Generation**	**TP**	**Tool**	**Spec. Language**	**Quiesc.**
Cardell-Oliver	offline	yes*	Essex*	TIOLTS	no
En-Nouaary et al.	offline	yes	no	deterministic and output urgent TAIO	no
Li et al.	offline	yes	no	RT Statecharts	no
Khoumsi	offline	yes	no	non-deterministic TIOSA	no
Briones and Brinksma	offline	no	no	TIOLTS	yes
Bohnenkamp and Belinfante	online	yes	yes*	non-deterministic safety TAIO	yes
Bodeveix et al.	offline	yes	no	a kind of TAIO	no
Larsen et al.	online	yes	TRON	TAIO (with guards on locations and transitions)	no
Hessel et al.	offline	yes	CoVer	deterministic and output urgent TAIO	no
Merayo et al.	offline	no	no	non-deterministic TEFSM	no
Krichen and Tripakis	offline and online	yes	TTG*	partially-observable and non-deterministic TAIO	no
Zheng et al.	offline	yes*	TROMLAB*	TEFSM	no
David et al.	offline	yes	TIGA	TIOGA	no
Adjir et al.	offline	yes	TINA	Prioritized Time Petri Nets	no
Styp et al.	no	no	no	STA	no
Timo et al.	offline	yes	no	VDTA	no

Table 4.1: Related Work

In the second table (Table 4.2), the following characteristics are considered: (1) definition of a conformance relation; (2) assumptions related to the specification; (3) assumptions related to the implementation under test. Considering the first characteristic, most of the approaches define a conformance relation based on either trace equivalence or trace inclusion. These kinds of conformance are very restricted because the implementation must have inputs and outputs defined in the specification. In the case where the specification does not completely specify a system, an implementation is allowed to have inputs not defined in the specification, thus only the outputs related to inputs defined in the specification must be considered. In this sense, Krichen and Tripakis (Subsection 4.1.11) propose a less restricted conformance relation, named tioco. Briones and Brinksma (Subsection 4.1.5) is the only work that defines a conformance relation considering quiescence and the idea is implemented by Bohnenkamp and Belinfante (Subsection 4.1.6). Merayo et al. (Subsection 4.1.10) define several conformance relations which can be considered as timed trace inclusions as well as the conformance relation defined by Timo et al. (Subsection 4.1.16). An interesting conformance relation is defined by Styp et al. (Subsection 4.1.15) based on symbolic constraints instead of concrete outputs, however at the semantic level it coincides with tioco.

Considering the second characteristic in Table 4.2, as already discussed, several approaches impose determinism to the specification in order to simplify the strategy. Almost all approaches assume input-completeness of the specification, so a complete specification of the IUT must be available. This restriction is relaxed by Krichen and Tripakis (Subsection 4.1.11) where the testing process can be performed with a system partially specified.

The last characteristic in Table 4.2 is related to the assumptions about the IUT. Practically, all approaches assume the input-completeness of the IUT. In practice, this assumption is true in many contexts, but not all. There are several scenarios where an IUT may not be input-complete, for instance, when a user tries to save a read-only text or to insert a PIN card in a slot of a cash machine where there is already another PIN card inserted. It is clear that there are situations where the inputs can be forbidden or ignored by the system. In this sense, Briones and Brinksma (Subsection 4.1.5) provide a strategy to deal with these cases.

In the third table (Table 4.3), the following characteristics are considered: (1) the kind of time (analogue or digital-time); (2) the kind of test cases (instantiated or abstract); (3) the kind of communication allowed by models; (4) the kind of oracle (manual, partial, or automated). Considering the first characteristic of Table 4.3, almost all approaches adopt the analogue-time model and represent time in a symbolic way. An interesting characteristic of Krichen and Tripakis' work is that they do not represent time symbolically, but they provide

Work	**Conf. Relation**	**Specification**	**Implementation**
Cardell-Oliver	trace equivalence	input-complete and must have more states than the implementation.	input-complete
En-Nouaary et al.	trace equivalence	input-complete and must have the same number of locations as the implementation.	input-complete
Li et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Khoumsi	timed trace inclusion	input-complete	input-complete
Briones and Brinksma	ioco with quiescence	input-complete	input-complete*
Bohnenkamp and Belinfante	ioco with quiescence	input-complete	input-complete
Bodeveix et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Larsen et al.	timed trace inclusion	deterministic and input-complete	input-complete
Hessel et al.	timed trace inclusion	deterministic, input-complete, and output urgent	input-complete
Merayo et al.	there are several conformance relations	input-complete	input-complete
Krichen and Tripakis	tioco	no restriction on input-completeness	input-complete
Zheng et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
David et al.	tioco	input-complete	input-complete
Adjir et al.	timed trace inclusion	deterministic, input-complete, and output urgent	deterministic, input-complete, and output urgent
Styp et al.	stioco	non-deterministic	input-complete
Timo et al.	tvco	assumptions are not discussed	assumptions are not discussed

Table 4.2: Related Work

a symbolic reachability algorithm to generate tests. Timo et al. (Subsection 4.1.16) propose a reachability analysis based on clock regions, but this strategy can quickly lead to the state space explosion problem. It is important to remark that complete real-time symbolic testing strategies that take system variables, parameters and time into account are nonexistent.

Almost all approaches generate instantiated test cases, since only time is abstracted during the test generation. Considering the approaches proposed by Styp et al. (Subsection 4.1.15) and Timo et al. (Subsection 4.1.16), it is not possible to define the kind of test case because neither examples are presented nor test cases are formally defined. Considering the possibility of specification of communicating elements, all approaches only allow synchronous communication. Thus, it is not possible to model asynchronous events such as interruptions. Considering the last characteristic in Table 4.3, most approaches only provided test case generation algorithms, but in the context of real-time systems the execution of test cases and verdicts assignment are as difficult as the generation of tests. Only three approaches developed an automated oracle. Basically, Bohnenkamp and Belinfante (Subsection 4.1.6), Larsen et al. (Subsection 4.1.8), and Krichen and Tripakis (Subsection 4.1.11) developed algorithms that use the specification to guide the execution of tests and assignment of verdicts. When the specification has only actions without parameters and variables the development of automated oracles is relatively simple. Nevertheless, an automated oracle in a real-time symbolic testing strategy causes the test data generation problem because variables and parameters must be instantiated during the test execution.

4.3 Problem Statements

This section describes several problems identified during the review of the work related to this thesis. A practical example adapted from [110] will be used to clarify the discussion. The chosen example is a burglar alarm system, a real-time monitoring system. The objective of the system is to monitor sensors to detect the presence of intruders in a building.

This system uses different kinds of sensors including movement detectors in individual rooms, window sensors, which detect the breaking of a window and door sensors, which detect the opening of doors. There are 50 window sensors, 30 door sensors, and 200 movement detectors. When a sensor indicates the presence of an intruder, the system automatically

Work	**Time**	**Test Cases**	**Communication**	**Oracle**
Cardell-Oliver	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
En-Nouaary et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Li et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Khoumsi	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Briones and Brinksma	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Bohnenkamp and Belinfante	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Bodeveix et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Larsen et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Hessel et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Merayo et al.	digital-time model	instantiated	synchronous	partial
Krichen and Tripakis	digital and analogue-time models*	instantiated	synchronous	automated
Zheng et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
David et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Adjir et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Styp et al.	analogue-time model	undefined	synchronous	undefined
Timo et al.	analogue-time model	undefined	synchronous	undefined

Table 4.3: Related Work

calls the police and, with a voice synthesiser, reports the position of the alarm. In addition, the system switches on lights around the area with activated sensors and switches on an audible alarm. The system is normally powered by the central power supply system, but it is equipped with a battery backup. The loss of power is detected by a circuit monitor that monitors the main tension. The system switches automatically to backup power when a voltage drop is detected. The timing requirements contained in our version of the burglar alarm system are described in Table 4.4.

Stimulus/Response	**Timing Requirements**
Power fail interrupt	The switch to backup power must be completed within a deadline of 50 ms.
Audible alarm	The audible alarm should be switched on within 1/2 second after an alarm is raised by a sensor.
Voice synthesiser	A synthesised message should be available within 3 seconds after an audible alarm is switched on.
Communications	The call to the police should be started within 1 second after a message is synthesised.
Lights switch	The lights should be switched on within 1/2 second after the calling to the police.

Table 4.4: Timing Requirements

Considering the architecture of the system, each system functionality is allocated to a concurrent process as well as each kind of sensor is allocated to a process. There is an interruption-driven system to deal with the failure and switching of power supply, a communication system, a voice synthesiser, an audible alarm system, and an illumination drive system to turn on lights around the sensor. The architecture of the system is depicted in Figure 4.1. The labelled arrows indicate the data flow between processes and the notes associated with the processes indicate which process or action causes the interruption.

Considering the specified scenario, some limitations of the approaches presented in this chapter are discussed. As an alarm system is naturally a system that has several quiescent states it is important to consider this property during the testing process. In this sense, the first problem that arises is the lack of tools for testing quiescence in real-time systems.

Another limitation of the current work is the input-completeness assumption of implementations. As discussed in Section 4.2, there are situations where some inputs can be forbidden or ignored by the system. These cases are only considered by Briones and Brinksma [24], but all forbidden or ignored inputs must be previously specified. Thus, there are still

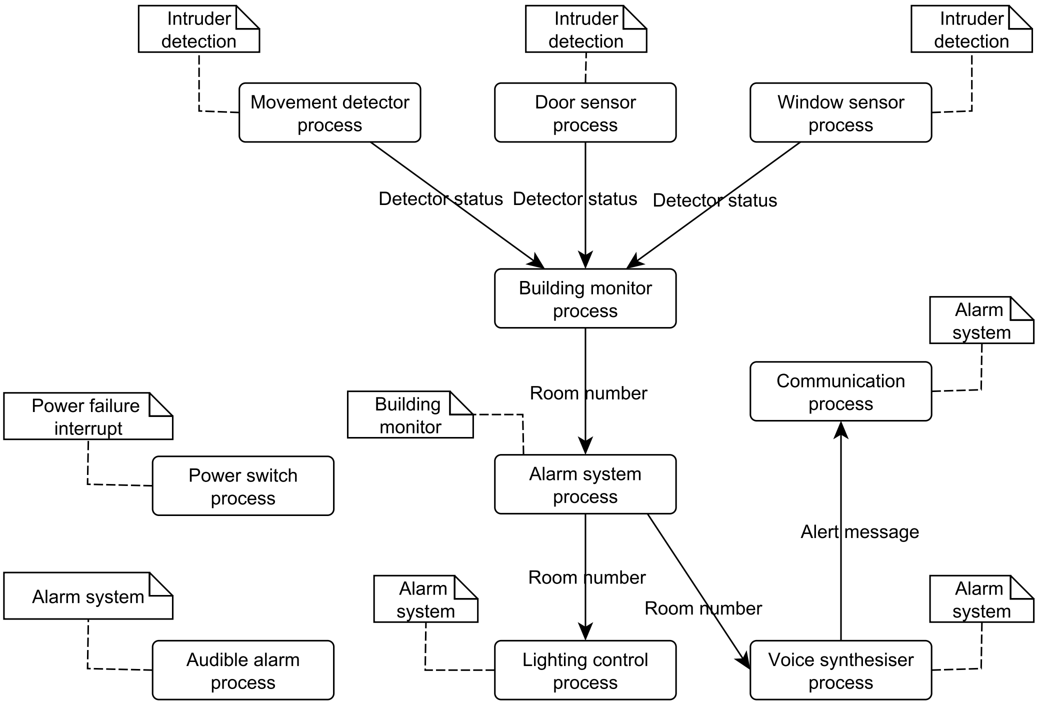


Figure 4.1: Burglar Alarm System Architecture

open problems such as the cases where a system cannot accept an input because of a fault (e.g. a requirement was implemented but there is not an option menu to access it). This problem is very common in the test of mobile phone applications and it is present not only in the real-time systems context but also in non-real-time systems.

The most efficient implementation of the burglar alarm system is to adopt the interrupt-driven architecture of Figure 4.1 where most communication between processes is asynchronous. For example, when an intruder is detected, the process that controls the activated sensors interrupts the building monitor process, which interrupts the alarm system process. Finally, the alarm system process, using interruptions, activates the following processes: audible alarm, lighting control, voice synthesiser, and communication process. However, current real-time models do not take asynchronous events into account. Thus, the testing process of real-time systems with asynchronous events based on formal models is compromised, if not impossible.

As discussed in Section 4.2, all approaches only use symbolic strategies to abstract time as a way of digitisation of analogue-time models. Nevertheless, when the system uses huge data domains, each value continues to be represented as a system state, leading to the classical state space explosion problem. Moreover, the digitisation of analogue-time models usually leads to a huge number of test cases. An interesting solution would be to provide symbolic testing strategies to abstract not only time but also variables of the system, leading to a simplified test suite where only most significant system data would be used during the test execution.

Taking the burglar alarm system as an example, the representation of the three kinds of sensors (movement, door and window sensors) using the existing real-time models would lead to three different paths in the model. In a real-time symbolic model, the representation is simple: the processes representing the three kinds of sensors could be abstracted in only one location in the model, where the interruption action would carry the information about the kind of sensor and the room number as parameters. In addition, the number of rooms of the burglar alarm system could be abstracted in a variable, thus it would be simpler to model situations where depending on the number of activated sensors, the system could take different decisions.

The high abstraction level in symbolic models leads to other problems such as the oracle problem. As it is possible to generate abstract test cases, the oracle problem is related to symbolic strategies in the sense that it is more difficult to provide an automated way for test case generation and execution, since test cases must be instantiated according to constraints defined in the specification.

4.4 Concluding Remarks

This chapter presented a review of work related to this thesis and several problems were stated. It is not our intention to deal with all identified problems. As discussed in Chapter 1, we intend to propose an extension of the symbolic testing strategy presented in Subsection 2.1.9 to deal with real-time systems. Furthermore, we intend to provide ways of modelling and testing asynchronous events considering an automated oracle for test case generation and execution. Problems related to quiescence and input-completeness are outside the scope of this thesis.

Chapter 5

Timed Input-Output Symbolic Transition Systems

This chapter presents a new symbolic model named Timed Input-Output Symbolic Transition System (TIOSTS) [11]. The goal is to address limitations of the existing notations and abstract time and data during test case generation. This model is an extension of two existing models: Timed Automata with Inputs and Outputs (TAIO), itself an extension of timed automata [5] with distinguished inputs and outputs, and deadlines to model urgency [21]; and Input-Output Symbolic Transition Systems (IOSTS) [108]. In other words, a TIOSTS is an automaton with a finite set of locations, variables used to represent the system data, and a finite set of clocks used to represent time evolution. An edge comprises a guard on variables and clocks, an action carrying parameters for the communication with its environment, an assignment to variables, and resets of clocks.

5.1 Syntax of TIOSTS

We intuitively explain the different notions of the TIOSTS model through the example shown in Figure 5.1[1] that models a withdrawal transaction in an ATM system. In a TIOSTS, a transition is fired if its guard is true, then the action is executed and all assignments are performed.

[1] In graphical representations, input actions are followed by the "?" symbol and output actions are followed by the "!" symbol. These symbols are used only as visual notation, they are not part of the action's name.

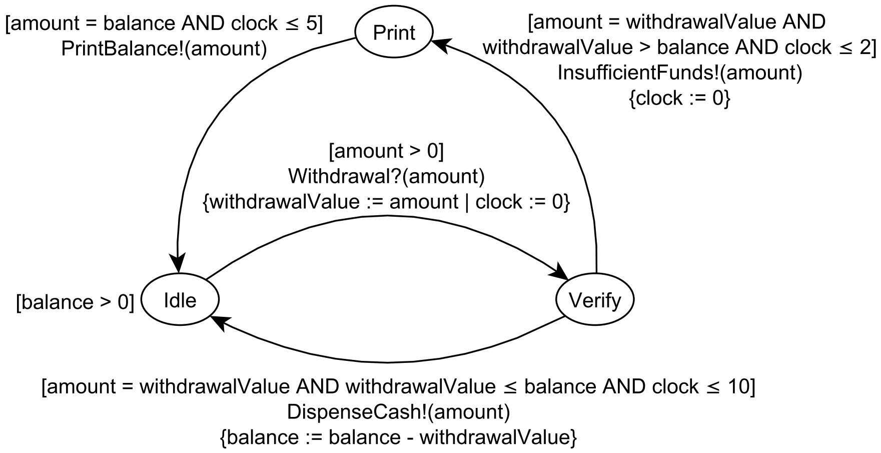
The withdrawal transaction has a precondition (an initial condition) that states that the current *balance* must be strictly positive. Initially, the system is in the *Idle* location where it expects the *Withdrawal* input carrying a strictly positive integer parameter *amount* that is saved into the *withdrawalValue* variable with the clock set to zero when the transition is taken. The scope of an action parameter is local with respect to the transition where it appears, thus the value of an action parameter must be stored in a variable in order to use it in the future.

Considering that the value of *withdrawalValue* is less than or equal to the *balance* and the time represented by *clock* is less than or equal to 10 time units, the ATM system dispenses the cash through the *DispenseCash* output carrying the *amount* parameter (the condition $amount = withdrawalValue$ contained in the guard means "choose a value for the *amount* parameter that, with the value of the *withdrawalValue* variable, satisfies the guard"). This is a characteristic inherited from IOSTS models whose objective is to associate the value of a variable with an action parameter in order to define output actions. Finally, the balance variable is decreased by the withdrawn value, and the system returns to the *Idle* location.

On the other hand, if the account does not have sufficient funds, the system must emit the invalid withdrawal value through the *InsufficientFunds* output carrying the *amount* parameter when *clock* is at most 2 (the condition $amount = withdrawalValue$ has a similar meaning to the previous guard), and reset the clock to zero again. Finally, the current balance is emitted through the *PrintBalance* output when *clock* is at most 5 (the condition $amount = balance$ contained in the guard means "choose a value for the *amount* parameter such that it is equal to the value of the *balance* parameter"), and the system returns to the *Idle* location.

Guards on transitions only indicate when they are enabled or not, but they cannot force the transition to be taken. Considering the specification of Figure 5.1, the TIOSTS may stay forever in any location. This can be solved by adding some restrictions to the transitions in order to describe the urgency of execution. Adopting the strategy defined in [21], each transition is annotated with one of the following three deadlines: *lazy*, *delayable*, and *eager*. The *lazy* deadline imposes no urgency to the transition to be taken, *delayable* means that once enabled the transition must be taken before it becomes disabled, and *eager* means the transition must be taken as soon as it becomes enabled.

Default deadlines are adopted in order to not overload pictures, thus when not specified



Print
[amount = balance AND clock ≤ 5]
PrintBalance!(amount)
[amount = withdrawalValue AND
withdrawalValue > balance AND clock ≤ 2]
InsufficientFunds!(amount)
{clock := 0}
[amount > 0]
Withdrawal?(amount)
{withdrawalValue := amount | clock := 0}
[balance > 0]
Idle
Verify
[amount = withdrawalValue AND withdrawalValue ≤ balance AND clock ≤ 10]
DispenseCash!(amount)
{balance := balance - withdrawalValue}

Figure 5.1: TIOSTS Example

the deadline of transitions with output actions is assumed to be *delayable* and the deadline of transitions with input actions is assumed to be *lazy*. On the other hand, when different deadlines are necessary they must be explicitly specified. A TIOSTS is formally described in Definition 5.1.

Definition 5.1 (TIOSTS)**.** *Formally, a TIOSTS is a tuple $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$, where:*

- *V is a finite set of typed variables;*
- *P is a finite set of parameters. For $x \in V \cup P$, $type(x)$ denotes the type of x;*
- *Θ is the initial condition, a predicate with variables in V;*
- *L is a finite, non-empty set of locations;*
- *$l^0 \in L$ is the initial location;*
- *$\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is a non-empty, finite alphabet, which is the disjoint union of a set $\Sigma^?$ of input actions, a set $\Sigma^!$ of output actions, and a set Σ^τ of internal actions. Each action $a \in \Sigma$ has a signature $sig(a) = \langle p_1, ..., p_n \rangle$, that is a tuple of distinct parameters. The signature of internal actions is the empty tuple;*
- *C is a finite set of clocks;*
- *$\mathcal{T}$ is a finite set of transitions. Each transition $t \in \mathcal{T}$ is a tuple $\langle l, a, G, A, y, l' \rangle$, where:*
  - *$l \in L$ is the origin location of the transition,*

- *$a \in \Sigma$ is the action,*
- *$G = G^D \wedge G^C$ is the guard, where G^D is a predicate over variables in $V \cup sig(a)$*[2] *and G^C is a clock constraint over C defined as a conjunction of constraints of the form $\alpha \# c$, where $\alpha \in C$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$,*
- *$A = A^D \cup A^C$ is the assignment of the transition. For each variable $x \in V$ there is exactly one assignment in A^D, of the form $x := A^{D^x}$, where A^{D^x} is an expression on $V \cup sig(a)$. $A^C \subseteq C$ is the set of clocks to be reset,*
- *$y \in \{$lazy, delayable, eager$\}$ is the deadline of the transition,*
- *$l' \in L$ is the destination location of the transition.*

◇

5.2 Semantics of TIOSTS

TLTS and TIOLTS models are used to define the semantics of all approaches based on TA and TAIO, respectively. Thus, the semantics of a TIOSTS $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ is described in terms of a TIOLTS (Definition 5.2). Intuitively, the TIOLTS states expand the sets of locations, of *valuations* of variables V and clocks C, while transitions expand the sets of actions Σ associated with parameter values P. A *valuation* of the variables in V is a mapping ν which maps every variable $x \in V$ to a value $\nu(x)$ in the domain of x. Valuations of parameters P are defined similarly. Let $\mathcal{V}$ denote the set of valuations of the variables V and let Γ denote the set of valuations of the parameters P. Let the function $\psi : C \rightarrow \mathbb{R}^{\geq 0}$ denote a clock valuation. We denote by $\overline{0}$ the valuation that assigns 0 to all clocks.

Considering $\nu \in \mathcal{V}$ and $\gamma \in \Gamma$, for an expression E involving a subset of $V \cup P$, we denote by $E(\nu, \gamma)$ the value obtained by evaluating the result of substituting in E each variable by its value according to ν and each parameter by its value according to γ.

Definition 5.2 (TIOLTS semantics of a TIOSTS)**.** *The semantics of a TIOSTS $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ is a TIOLTS $[\![W]\!] = \langle S, S^0, Act, T \rangle$, defined as follows:*

- *$S = L \times \mathcal{V} \times (C \rightarrow \mathbb{R}^{\geq 0})$ is the set of states of the form $s = \langle l, \nu, \psi \rangle$ where $l \in L$ is a location, $\nu \in \mathcal{V}$ is a specific valuation for all variables V, and ψ is a clock valuation;*

[2] G^D is assumed to be expressed in a theory in which satisfiability is decidable.

- *$S^0 = \{\langle l^0, \nu, \psi\rangle \mid \Theta(\nu) = true,\ \overline{0}\}$ is the set of initial states. It is important to remark that the number of initial states can be infinite because, in this case, there may be infinite valuations satisfying the initial condition Θ;*

- *$Act = \Lambda \cup D$ is the set of actions, where $\Lambda = \{\langle a, \gamma\rangle \mid a \in \Sigma, \gamma \in \Gamma_{sig(a)}\}$ is the set of discrete actions and $D = \mathbb{R}^{\geq 0}$ is the set of time-elapsing actions. Λ is partitioned into the sets $\Lambda^?$ of input actions, $\Lambda^!$ of output actions, and Λ^τ of internal actions;*

- *T is the transition relation defined as follows: (1) transitions with discrete actions are of the form $\langle l, \nu, \psi\rangle \overset{\langle a,\gamma\rangle}{\rightarrow} (l', \nu', \psi')$, where the system moves from $\langle l, \nu, \psi\rangle$ to $\langle l', \nu', \psi'\rangle$ through an action $\langle a, \gamma\rangle$ if there is a transition $t : \langle l, a, G, A, y, l'\rangle \in \mathcal{T}$ such that G evaluates to $true$, $\nu' = A^D(\nu, \gamma)$, and $\psi' = A^C(\psi)$; (2) transitions with time-elapsing actions are of the form $(l, \nu, \psi) \overset{d}{\rightarrow} (l, \nu, \psi + d)$ for all $d \in D$ considering that the deadlines do not block time progress. Once the lazy deadline is used only to denote the absence of deadlines, lazy transitions cannot block time progress. A delayable transition can block time progress if there exist $0 \leq d_1 < d_2 \leq d$ such that $\psi + d_1 \models G^C$ and $\psi + d_2 \not\models G^C$, whereas an eager transition can block time progress if $\psi \models G^C$.*

◇

As in [78], delayable transitions with guards of the form $\alpha < c$ are not allowed because there is no latest time so that the guard is still true. Also, eager transitions with guards of the form $\alpha > c$ are not allowed because there is no earliest time so that the guard becomes true.

Most notions and properties of TIOSTS are defined in terms of their underlying TIOLTS semantics (Definition 5.2). Then, consider $s, s', s_i \in S$; $\tau_i \in \Lambda^\tau$; $\omega, \omega_i \in Act$; and $a, a_i \in (Act\backslash\Lambda^\tau)$. Moreover, let $\rho \in Act^*$ be a sequence of discrete actions and time-elapsing actions, and $\sigma \in (Act\backslash\Lambda^\tau)^*$ be a sequence of visible discrete and time-elapsing actions. $\epsilon \in Act^*$ is the empty sequence. The sum of all delays spent in a sequence of actions ρ (respectively σ) is denoted by time(ρ) (respectively by time(σ)). For example, time(ϵ) $= 0$ and time($2.5\ a?\ 0.5\ x!$) $= 3.0$.

Let $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$ be a TIOSTS whose semantics is defined by the TIOLTS $[\![W]\!] = \langle S, S^0, Act, T\rangle$. We write $s \overset{\omega}{\rightarrow} s'$ for $(s, w, s') \in T$, $s \overset{\omega}{\rightarrow}$ for $\exists s' : s \overset{\omega}{\rightarrow} s'$. Let $s \overset{\omega_1 \ldots \omega_n}{\rightarrow} s' \overset{\Delta}{=} \exists s_0, ..., s_n : s = s_0 \overset{\omega_1}{\rightarrow} s_1 \overset{\omega_2}{\rightarrow} \cdots \overset{\omega_n}{\rightarrow} s_n = s'$ be an *execution*. We also write

$s \xrightarrow{\rho}$ for $\exists s' : s \xrightarrow{\rho} s'$. *Traces*$(s) \triangleq \{\rho \in Act^* \mid s \xrightarrow{\rho}\}$ describes the set of sequences of discrete and time-elapsing actions fireable from s. The set of fireable actions from s is defined by $\Omega(s) \triangleq \{\omega \in Act \mid s \xrightarrow{\omega}\}$. $Out(s) \triangleq \Omega(s) \cap (\Lambda^! \cup D)$ is the set of all output events (including time-elapsing actions) fireable from s. The definition of $Out(s)$ can be extended for sets of states: for $P \subseteq S$ we have $Out(P) \triangleq \bigcup_{s \in P} Out(s)$.

The $\Rightarrow$ relation is used to denote the observable behaviour. Given $s, s' \in S$, $d \in \mathbb{R}^{\geq 0}$ and $a \in \Lambda^! \cup \Lambda^?$, we have $s \overset{d}{\Rightarrow} s'$ whenever $\exists \rho \in (\Lambda^\tau \cup D)^*$ such that $s \xrightarrow{\rho} s'$ and $time(\rho) = d$, whereas we have $s \overset{a}{\Rightarrow} s'$ whenever $\exists \rho_1, \rho_2 \in (\Lambda^\tau)^*$, $s_1, s_2 \in S$ such that $s \xrightarrow{\rho_1} s_1 \xrightarrow{a} s_2 \xrightarrow{\rho_2} s'$. Given $a_1, \cdots a_n \in (Act \backslash \Lambda^\tau)^*$, an *observable execution* is defined as $s \overset{a_1 \ldots a_n}{\Rightarrow} s' \triangleq \exists s_0, ..., s_n :$ $s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} s_n = s'$. For $a \in Act \setminus \Lambda^\tau$ we also define $s \overset{a}{\Rightarrow} \triangleq \exists s' : s \overset{a}{\Rightarrow} s'$ and for $\sigma \in (Act \setminus \Lambda^\tau)^*$, $s \overset{\sigma}{\Rightarrow} \triangleq \exists s' : s \overset{\sigma}{\Rightarrow} s'$. *ObservableTraces*$(s) \triangleq \{\sigma \in (Act \setminus \Lambda^\tau)^* \mid s \overset{\sigma}{\Rightarrow}\}$ describes the set of sequences of observable and time-elapsing actions fireable from s. Finally, the set of sequences of observable behaviours fireable from the initial state of a TIOSTS W is defined by *ObservableTraces*$(W) \triangleq$ *ObservableTraces*(S_0).

The set s *after* $\sigma \triangleq \{s' \in S \mid s \overset{\sigma}{\Rightarrow} s'\}$ is the set of states reachable from s after the execution of σ, and P *after* $\sigma \triangleq \bigcup_{s \in P} s$ *after* σ is the set of states reachable from the set P after the execution of σ.

Subclasses of TIOSTS. Let $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ be a TIOSTS and $[\![W]\!] = \langle S, S^0, Act, T \rangle$ its associated TIOLTS. W is *complete* if it can accept any action at any state, i.e., $\forall s \in S, b \in \Lambda : s \xrightarrow{b}$. On the other hand, W is *input-complete* if it can accept any input action at any state, possibly after internal actions, i.e., $\forall s \in S, b \in \Lambda^? : s \overset{b}{\Rightarrow}$. W is said to be a *lazy-action* TIOSTS if the deadlines of all transitions are lazy, that is, $\mathcal{T} = \{t \mid t : \langle l, a, G, A, lazy, l' \rangle \in \mathcal{T}\}$. W is said to be a *non-blocking* TIOSTS when it does not block time. In this case, the following condition must be satisfied [74]: $\forall s \in \; S_0$ *after* ρ, $\forall d \in \mathbb{R}^{\geq 0}$, $\exists \rho' \in (\Lambda^! \cup \Lambda^\tau \cup D)^* \; : \; \text{time}(\rho') = d \wedge s \xrightarrow{\rho'}$. W is said to be *deterministic* if the following three conditions are satisfied [35; 63; 108]:

1. $\Lambda^\tau = \emptyset$ (i.e. there are no internal actions);

2. $\mid S_0 \mid = 1$, that is, there is only one initial state implying the initial condition Θ is satisfied by only one valuation ν^0;

3. for all $l \in L$ and for each pair of distinct transitions with origin in l carrying the same action a, that is, $t_1 : \langle l, a, G_1, A_1, y_1, l'_1 \rangle$ and $t_2 : \langle l, a, G_2, A_2, y_2, l'_2 \rangle$, the guards G_1 and G_2 are mutually exclusive (i.e., $G_1 \wedge G_2$ is unsatisfiable).

5.3 Synchronous Product of TIOSTS

The synchronous product of two TIOSTSs W_1 and W_2 is an important operation used in both property oriented testing and conformance testing. This operation is used in the former for identifying behaviours of the specification accepted or rejected by a particular property (e.g., W_1 could be a specification and W_2 could be a test purpose). On the other hand, for conformance testing, this operation is used for modelling the synchronous execution of a test case on an implementation (e.g., W_1 could be a test case and W_2 could be an implementation under test). This classical problem is known as the language intersection problem [62].

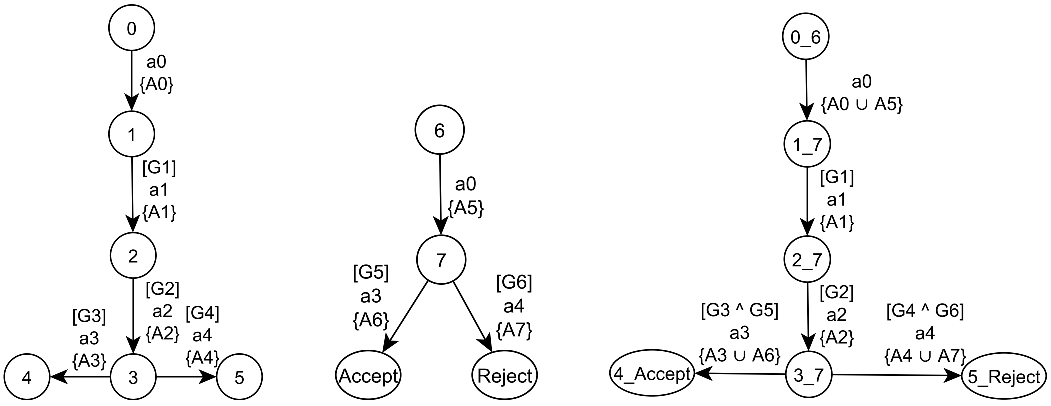
The synchronous product operation requires compatibility between W_1 and W_2, that is, W_1 and W_2 must share the same sets of input and output actions from the same signature, with the same set of parameters, and have no variables, internal actions, or clocks in common.

Definition 5.3 (Compatibility for Synchronous Product)**.** *The TIOSTSs* $W_i = \langle V_i, P_i, \Theta_i, L_i, l_i^0, \Sigma_i, C_i, \mathcal{T}_i \rangle$ $(i = 1, 2)$ *are compatible if* $V_1 \cap V_2 = \emptyset, P_1 = P_2, \Sigma_1^? = \Sigma_2^?, \Sigma_1^! = \Sigma_2^!, \Sigma_1^\tau \cap \Sigma_2^\tau = \emptyset$, *and* $C_1 \cap C_2 = \emptyset$. ◇

Given the ordering $lazy < delayable < eager$ on deadlines and two deadlines y_1, y_2, $op(y_1, y_2) = (y_2$ if $y_1 < y_2$ and y_1 otherwise$)$ is an operation which computes the resulting deadline in the synchronous product operation by keeping the most restrictive one.

Given two compatible TIOSTSs, Definition 5.4 formally describes the synchronous product between them.

Definition 5.4 (Synchronous Product)**.** *The synchronous product of two compatible TIOSTSs* W_1 *and* W_2 *is denoted by* $SP = W_1 \parallel W_2$. SP *is the TIOSTS* $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ *defined by:* $V = V_1 \cup V_2, P = P_1 = P_2, \Theta = \Theta_1 \wedge \Theta_2, L = L_1 \times L_2, l^0 = \langle l_1^0, l_2^0 \rangle, \Sigma^? = \Sigma_1^? = \Sigma_2^?, \Sigma^! = \Sigma_1^! = \Sigma_2^!, \Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$, *and* $C = C_1 \cup C_2$. *The set* $\mathcal{T}$ *is the smallest set such that:*



(a) TIOSTS W_1 (b) TIOSTS W_2 (c) $SP = W_1 \parallel W_2$

Figure 5.2: Synchronous Product Example

1. *For* $a \in \Sigma_1^\tau$ *and* $l_2 \in L_2$*:*

$$\textit{if } \langle l_1, a, G_1, A_1, y_1, l_1' \rangle \in \mathcal{T}_1 \;\; \textit{then } \langle \langle l_1, l_2 \rangle, a, G_1, A_1, y_1, \langle l_1', l_2 \rangle \rangle \in \mathcal{T};$$

2. *For* $a \in \Sigma_2^\tau$ *and* $l_1 \in L_1$*:*

$$\textit{if } \langle l_2, a, G_2, A_2, y_2, l_2' \rangle \in \mathcal{T}_2 \;\; \textit{then } \langle \langle l_1, l_2 \rangle, a, G_2, A_2, y_2, \langle l_1, l_2' \rangle \rangle \in \mathcal{T};$$

3. *For* $a \in \Sigma^? \cup \Sigma^!$*:*

$$\textit{if } \langle l_1, a, G_1, A_1, y_1, l_1' \rangle \in \mathcal{T}_1 \;\; \textit{and } \langle l_2, a, G_2, A_2, y_2, l_2' \rangle \in \mathcal{T}_2 \;\; \textit{then}$$

$$\langle \langle l_1, l_2 \rangle, a, G_1 \wedge G_2, A_1 \cup A_2, op(y_1, y_2), \langle l_1', l_2' \rangle \rangle \in \mathcal{T}.$$

◇

Considering the Definition 5.4, the execution of internal actions can occur independently and it is described by Rules 1 and 2. Rule 3 describes the synchronization of W_1 and W_2 through observable actions. Figure 5.2 presents an example of the synchronous product between a TIOSTS W_1 (Figure 5.2(a)) and a TIOSTS W_2 (Figure 5.2(b)), obtaining the TIOSTS of Figure 5.2(c) as result.

5.4 Concluding Remarks

This chapter presented the symbolic model proposed to address limitations of the existing notations abstracting time and data in the specification of real-time systems. As the proposed model is based on timed automata, it has the same expressiveness as the approaches presented in Chapter 4. However, the use of variables to represent the system data leads to more compact and abstract models. Moreover, when data and time are treated in a symbolic way the state space explosion problem is avoided. Next chapter presents how test cases can be generated from the proposed model.

Chapter 6

Conformance Testing with TIOSTS

This chapter instantiates the conformance testing framework presented in Subsection 2.1.5 and the test purpose framework presented in Subsection 2.1.6 considering TIOSTS models defined in Chapter 5. After that, the whole process defined to generate test cases based on TIOSTS models in order to check the conformance between a specification and an implementation is described. Finally, some properties of the test cases generated by our approach are discussed. The contents of this chapter is also discussed in [11].

6.1 Testing Conformance

Conformance testing is a kind of testing used to ensure that an implementation of a software system meets its specification [118]. This kind of testing relates a specification with an implementation through a conformance relation, which is checked by the execution of test cases, possibly selected according to a test purpose. Thus, it is essential to describe all concepts related to conformance testing such as specifications, implementations, conformance relations between specifications and implementations, and test cases.

Specifications. A specification is a formal model of the SUT represented by a non-blocking TIOSTS $\mathcal{S}$. The non-blocking specification assumption is due to the fact that we are considering specifications of software systems that do not force input actions, that is, the system cannot block because an input action was not provided by the environment.

Implementations. An implementation is a physical software system running on a real-time environment (e.g., a real-time operating system). In order to reason about conformance, it is assumed that the semantics of any implementation can be modelled by a formal object. We assume here that it is modelled by a TIOLTS $\mathcal{I}$. Moreover, the implementation is assumed to be input-complete, non-blocking, and has the same interface (input and output actions with their signatures) as the specification $\mathcal{S}$. These assumptions are called *test hypotheses*.

Test Cases. Test cases (Definition 6.1) are used to check the conformance between specifications and implementations. It is here defined as a TIOSTS TC as follows:

Definition 6.1 (Test Case)**.** *A test case is a deterministic, input-complete TIOSTS* $TC = \langle V_{TC}, P_{TC}, \Theta_{TC}, L_{TC}, l^0_{TC}, \Sigma_{TC}, C_{TC}, \mathcal{T}_{TC} \rangle$*, equipped with three disjoint sets of locations* Pass*,* Fail*, and* Inconclusive*. Moreover, the set of actions is* $\Sigma_{TC} = \Sigma^?_{TC} \cup \Sigma^!_{TC}$*, where* $\Sigma^?_{TC} = \Sigma^!_{\text{SUT}}$ *(outputs of the* SUT *are the inputs of the TC) and* $\Sigma^!_{TC} = \Sigma^?_{\text{SUT}}$ *(TC emits only inputs allowed by the* SUT*).* ◇

Intuitively, when the location $Fail$ is reached, it means rejection, the location $Pass$ means that some targeted behaviour has been reached (this will be clarified later) and $Inconclusive$ means that targeted behaviours cannot be reached anymore.

Conformance Relation. The conformance relation considered is the **tioco** relation defined by Krichen and Tripakis in [76; 77]. Informally, an implementation conforms to a specification for **tioco** if and only if, after any trace of the specification, any output action (including time-elapsing actions) that the implementation provides after this trace is an output action that the specification may also provide.

Definition 6.2 (tioco)**.** *An implementation* $\mathcal{I}$ *conforms to a specification* $\mathcal{S}$ *for* ***tioco****, denoted by* $\mathcal{I}$ ***tioco*** $\mathcal{S}$*, iff* $\forall \sigma \in \text{ObservableTraces}(\mathcal{S}),\ Out(\mathcal{I}\ \textit{after}\ \sigma) \subseteq Out(\mathcal{S}\ \textit{after}\ \sigma)$. ◇

6.2 Test Case Generation Process

The test case generation process derives test cases from specifications according to the conformance relation. For simplicity, we shall assume that the specification $\mathcal{S}$ is deterministic and non-blocking. However, it is possible to deal with non-determinism, under some assumptions, for both data [65] and time [17]. It is important to remark that internal actions, quiescence, and non-input-completeness of implementations are not considered in the proposed test case generation process because these characteristics are outside the scope of this thesis. The proposed process considers the selection of test cases by test purposes.

Test Purposes. A test purpose describes some desired behaviours that we wish to check on the implementation during the test campaign. They are used to select test cases in order to check specific scenarios. In our setting, a test purpose is a particular TIOSTS TP formally described as follows:

Definition 6.3 (Test Purpose)**.** *Given a specification TIOSTS $\mathcal{S}$ with action alphabet Σ, a test purpose is a deterministic, complete, lazy-action TIOSTS $TP = \langle V_{TP}, P_{TP}, \Theta_{TP}, L_{TP}, l^0_{TP}, \Sigma_{TP}, C_{TP}, \mathcal{T}_{TP}\rangle$, equipped with a special set of locations* Accept $\subseteq L_{TP}$ *such that all transitions leaving these locations are self-loops*[1]. *Moreover TP has to be compatible with $\mathcal{S}$ thus $\Sigma_{TP} = \Sigma$.* ◇

The selection is performed through the synchronous product operation defined in Section 5.3. For this, complete test purposes are needed to ensure that the runs of a specification are not restricted before they are accepted (if ever).

Accept locations are used to indicate that the expected scenario modelled by the test purpose has been fulfilled, while *Reject* locations are used otherwise. Figure 6.1 presents an example of a test purpose for the withdrawal transaction example presented in Subsection 5.1. It is used to select the scenarios where the user successfully performs a withdrawal transaction. The *Reject* location is used to discard all other scenarios where the system does not exhibit the desired behaviour.

It is important to note that this test purpose is not complete (i.e., not all actions are enabled at any location), but using a strategy defined in [108] it is possible to automatically complete it. The steps to automatically complete a test purpose are: (1) in each location, add a self-loop with an action not enabled; (2) for each transition with a guard G and an action a, create a new transition to the *Reject* location with the same action a and the negation of the conjunction of all guards associated with a. With this automatic operation the activity of defining test purposes is simplified by allowing the tester to focus only on the desired

[1] One can also consider another set of locations $Reject$ that can be used to discard all other scenarios where the system does not exhibit the desired behaviour.

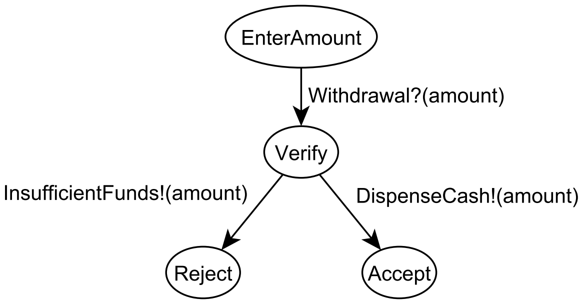


Figure 6.1: TIOSTS Test Purpose Example

behaviour.

The test case generation process starts with the specification $\mathcal{S}$ of the SUT $\mathcal{I}$ and a test purpose TP. Test purposes are used in order to verify if the SUT exhibits a desired behaviour and their definition allow the tester to focus only on specific behaviours. In this case, test purposes need to be completed because all input actions are not enabled all the time. The specification of $\mathcal{I}$ is combined with the completed test purpose through the computation of the synchronous product (Definition 5.4). Then, the resulting TIOSTS model is symbolically executed to identify and select possible traces leading to an *Accept* location. Finally, the selected trace is translated into a test case considering the TIOSTS notation. A general view of this test process is presented in Figure 6.2. Each step of this process is detailed in the remainder of this section.

6.2.1 Test Purpose Completion

Algorithm 6.1 presents a simplified implementation of the test purpose completion operation. This algorithm requires only one parameter: the test purpose to be completed.

During the algorithm execution all non-verdict locations of the test purpose are analysed (Lines from 2 to 4). If some location has not enabled actions, then a self loop is created with these actions (Lines from 5 to 14).

After that, for each outgoing transition of the location being processed that has any guard, it is created a new transition without assignments and as guard the negation of the conjunction of all guards associated with the action of the current transition (Lines from 15 to 22). If the target location of the transition being processed is the *Reject* location, then a self loop is created with this new transition (Line 24); otherwise, the target location of this new transition

Algorithm 6.1: Test Purpose Completion Algorithm

```
complete(TIOSTS $TP$){
  Set locations := $TP$.getLocations();
  for (Location location : locations) {
    if (!isVerdict(location)) {
      for (Action action : remaningActions(location)) {
        Transition transition := new Transition();
        transition.setSource(location);
        transition.setGuard(true);
        transition.setAction(action);
        transition.setAssignments($\emptyset$);
        transition.setDeadline(lazy);
        transition.setTarget(location);
        $TP$.addTransition(transition);
      }
      for (Transition t : location.getOutGoingTransitions()) {
        if (!t.getGuard().isEmpty()) {
          Transition transition := new Transition();
          transition.setSource(location);
          transition.setGuard(
              negation(location.getAllGuards(t.getAction())));
          transition.setAction(t.getAction());
          transition.setAssignments($\emptyset$);
          transition.setDeadline(lazy);
          if (t.getTarget() = reject) {
            transition.setTarget(location);
          } else {
            transition.setTarget(reject);
          }
          if (!$TP$.contains(transition)) {
            $TP$.addTransition(transition);
          }
        }
      }
    }
  }
}
```

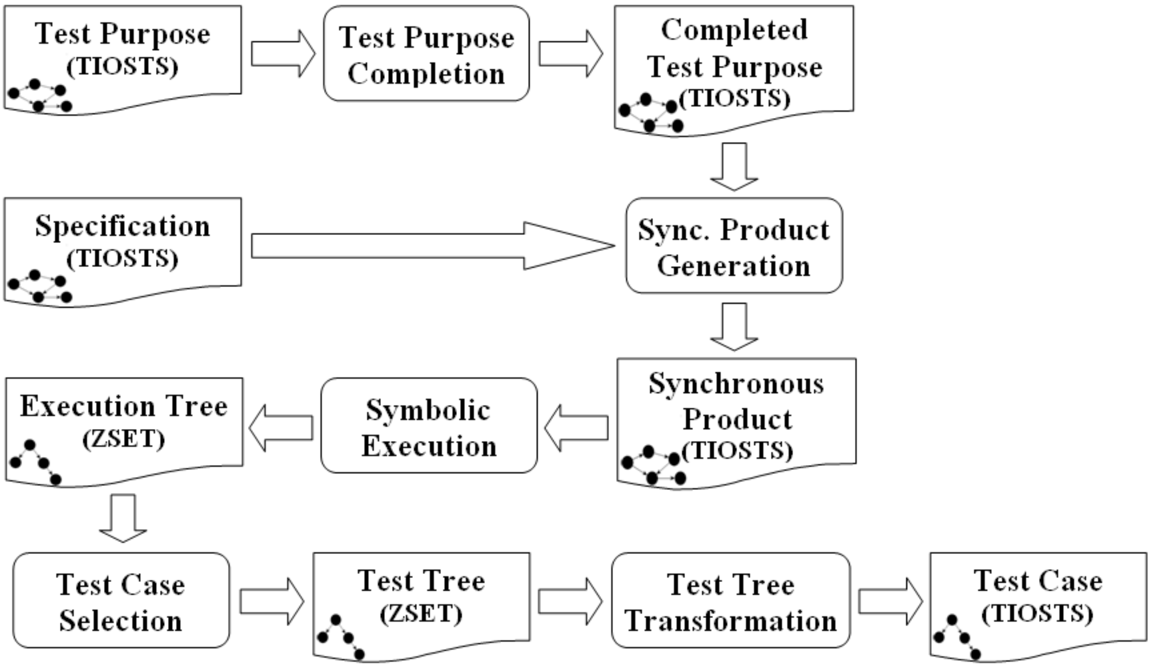


Figure 6.2: Test Case Generation Process

is set to the *Reject* location (Line 26). Finally, the created transition is added to the test purpose if it has not been added (Lines from 28 to 30).

Using the asymptotic notation, the running time of Algorithm 6.1 is $O(\mid L_{TP} \mid \cdot \mid \Sigma_{TP} \mid)$, where $\mid L_{TP} \mid$ is the number of locations of the test purpose and $\mid \Sigma_{TP} \mid$ is the size of the alphabet of the test purpose.

Figure 6.3 shows the completed test purpose generated by Algorithm 6.1 using, as parameter, the test purpose of Figure 6.1.

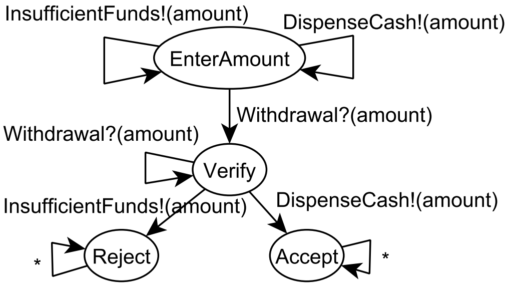


Figure 6.3: Completed Test Purpose Example

Algorithm 6.2: Synchronous Product of W_1 and W_2

```
1 synchronousProduct(TIOSTS $W_1$, TIOSTS $W_2$, TIOSTS syncProduct){
2   if(isCompatible($W_1$, $W_2$)){
3     product($l_1^0$, $l_2^0$, syncProduct);
4     mirror(syncProduct);
5   }
6 }
```

6.2.2 Synchronous Product Generation

Algorithms 6.2 and 6.3 present a simplified implementation of the synchronous product operation defined in Section 5.3. Algorithm 6.2 requires three TIOSTS as parameters: the synchronous product is computed between W_1 (specification) and W_2 (completed test purpose), and the result is returned in *syncProduct*.

Firstly, it is necessary to check whether W_1 and W_2 are compatible for synchronous product (Algorithm 6.2, Line 2) according to Definition 5.3. Then, the *product* method is used to traverse both TIOSTS from their initial locations following the Depth-First Search (DFS) strategy (Algorithm 6.2, Line 3).

Once the synchronous product is computed, the last action of Algorithm 6.2 (Line 4) is to invert input and output actions, in other words, all input actions become output actions and all output actions become input actions. This is important because during the test case execution the inputs of the SUT are outputs of the environment (tester) and vice-versa.

The *product* method is detailed in Algorithm 6.3. Three parameters are needed: l_1, the current location of W_1 being processed; l_2, the current location of W_2 being processed; and *syncProduct*, the synchronous product being computed. The first step (Algorithm 6.3, Line 2) is to check whether l_2 is a verdict location (Accept or Reject location). If so, the processing is stopped.

The loop in Line 5 of Algorithm 6.3 processes each transition t_1 leaving l_1 (i.e., the current location of W_1). $\mathcal{T}_2$ contains all transitions leaving the current location of W_2 that have the same action as t_1 (Algorithm 6.3, Line 6).

When $\mathcal{T}_2$ is empty (Algorithm 6.3, Line 9) means that there is no transition leaving the current location of W_2 with the same action as t_1. In this case, the new transition t of the

Algorithm 6.3: Product of W_1 and W_2

```
product(Location $l_1$, Location $l_2$, TIOSTS syncProduct) {
  if(isVerdict($l_2$)) {
    return;
  }
  for (Transition $t_1$ : $l_1$.getOutGoingTransitions()) {
    Set $\mathcal{T}_2$ := getTransitionsByAction($l_2$, $t_1$.getAction());
    Location source := new Location($l_1$.getLabel() + "_" + $l_2$.getLabel());
    Transition t := new Transition();
    if($\mathcal{T}_2$.isEmpty()){
      t.setSource(source);
      t.setGuard($t_1$.getGuard());
      t.setAction($t_1$.getAction());
      t.setDeadline($t_1$.getDeadline());
      t.setAssignments($t_1$.getAssignments());
      t.setTarget($t_1$.getTarget().getLabel + "_" + $l_2$.getLabel());
      if(!syncProduct.containsTransition(t)) {
        syncProduct.addTransition(t);
        product($t_1$.getTarget(), $l_2$, syncProduct);
      }
    } else {
      for (Transition $t_2$ : $\mathcal{T}_2$) {
        t.setSource(source);
        t.setGuard($t_1$.getGuard() + "AND" + $t_2$.getGuard());
        t.setAction($t_1$.getAction());
        t.setDeadline($t_1$.getDeadline());
        t.setAssignments($t_1$.getAssignments() + $t_2$.getAssignments());
        t.setTarget($t_1$.getTarget().getLabel() + "_" +
                                  $t_2$.getTarget().getLabel());
        if(!syncProduct.containsTransition(t)) {
          syncProduct.addTransition(t);
          product($t_1$.getTarget(), $t_2$.getTarget(), syncProduct);
        }
      }
    }
  }
}
```

synchronous product will be identical to t_1 (Lines from 11 to 14) excepting the source (Lines 7 and 10) and target (Line 15) locations. The test of Line 16 is necessary to avoid the inclusion of a transition twice when there are loops in the models and to guarantee that the algorithm terminates. If t has not been added to *syncProduct*, it is added and the algorithm continues recursively with the following parameters: l_1 becomes the target location of t_1, the same l_2 being processed, and *syncProduct*.

If $\mathcal{T}_2$ has one or more transitions with the same action as t_1, each transition $t_2 \in \mathcal{T}_2$ leads to creation of a new transition t to be added to *syncProduct* with the following properties: (1) the source location is the composition of the current locations (Lines 7 and 22); (2) the guard is the conjunction of the guards of t_1 and t_2 (Line 23); (3) the action is the same as t_1 (Line 24); (4) the deadline is the same as t_1 (Line 25); (5) the set of assignments is the union of the assignments set of t_1 and t_2 (Line 26); (6) the target location is the composition of target location of t_1 and target location of t_2 (Line 27).

If t has not been added to *syncProduct*, it is added and the algorithm continues recursively with target locations of t_1 and t_2, and *syncProduct* as parameters (Algorithm 6.3, Lines from 28 to 30).

Algorithms 6.2 and 6.3 have the same running time: $O(\mid \mathcal{T}_1 \mid + \mid \mathcal{T}_2 \mid)$, where $\mid \mathcal{T}_1 \mid$ is the number of transitions of the TIOSTS W_1 and $\mid \mathcal{T}_2 \mid$ is the number of transitions of the TIOSTS W_2.

Figure 6.4 shows the synchronous product generated by Algorithms 6.2 and 6.3 from specification of Figure 5.1 and completed test purpose of Figure 6.3.

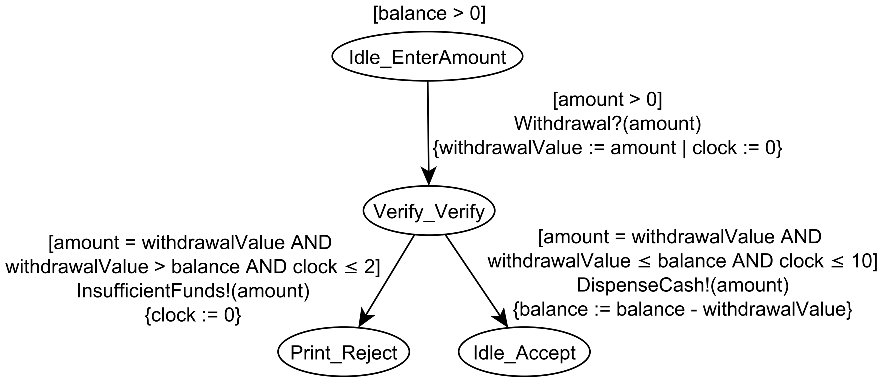

[balance > 0]
Idle_EnterAmount
[amount > 0]
Withdrawal?(amount)
{withdrawalValue := amount | clock := 0}
Verify_Verify
[amount = withdrawalValue AND
withdrawalValue > balance AND clock ≤ 2]
InsufficientFunds!(amount)
{clock := 0}
[amount = withdrawalValue AND
withdrawalValue ≤ balance AND clock ≤ 10]
DispenseCash!(amount)
{balance := balance - withdrawalValue}
Print_Reject
Idle_Accept

Figure 6.4: Synchronous Product Example

6.2.3 Symbolic Execution

Symbolic execution is a technique for analysing programs based on symbolic values as input rather than concrete values [33; 71]. Symbolic execution techniques were used by Gaston *et al.* [53] and Jöbstl *et al.* [66] for test generation for untimed systems. We here extend the work proposed by Jöbstl *et al.* [66] to deal with TIOSTS models.

The main idea is to symbolically execute TIOSTS models using the same technique used for symbolically executing programs. Thus, all possible traces are identified using symbolic values instead of concrete values for action parameters and variables of the model, avoiding the state space explosion problem w.r.t. the data part since data values are not enumerated. The resulting traces are represented as a zone-based symbolic execution tree (Definition 6.6), whose nodes are zone-based symbolic extended states (Definition 6.4) and edges are symbolic actions (Definition 6.5).

Definition 6.4 (Zone-Based Symbolic Extended State)**.** *A zone-based symbolic extended state (ZSES) of a TIOSTS* $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ *is a tuple* $\eta = \langle l, \pi, \varphi, Z \rangle$*, where:*

- $l \in L$ *is a location of* W*;*
- π *is the path condition, that is, a Boolean expression representing a data guard;*
- φ *is a mapping from variables and action parameters to their symbolic values;*
- Z *is a zone representing the solution set of a clock constraint.*

◇

Symbolically executing a TIOSTS implies that data and time must be taken into account. As in [66], path conditions are checked using constraint solving. However, our definition of states differs from [66] because zones are used to check the reachability of states w.r.t. time requirements: a state is reachable if its path condition π is satisfiable and its zone Z is not empty. Zones provide an efficient symbolic representation of time requirements, avoiding the state space explosion problem w.r.t. the time part. Furthermore, ZSESs are connected through transitions labelled by symbolic actions (Definition 6.5).

Definition 6.5 (Symbolic Action)**.** *A symbolic action is a tuple* $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$*, where:*

- $a \in \Sigma$ *is the corresponding action in the TIOSTS;*

- μ_{sa} *is a list of unique identifiers denoting the action parameters of* sa*;*

- φ_{sa} *is a mapping from the original action parameter names to the unique identifiers in* μ_{sa}.

◇

We are now ready to define zone-based symbolic execution trees:

Definition 6.6 (Zone-Based Symbolic Execution Tree)**.** *A zone-based symbolic execution tree (ZSET) is a deterministic, connected, acyclic graph represented by a tuple* $\langle S, SA, \eta^0, T\rangle$*, where:*

- S *is a finite set of zone-based symbolic extended states;*

- *SA is a finite set of symbolic actions;*

- $\eta^0 \in S$ *is the initial zone-based symbolic extended state;*

- T *is a finite set of transitions. Each transition* $t \in T$ *is a tuple* $\langle \eta, sa, \eta' \rangle$*, where:*
  - $\eta \in S$ *is the origin state of the transition,*
  - $sa \in$ *SA is the symbolic action of the transition,*
  - $\eta' \in S$ *is is the destination state of the transition.*

◇

A ZSET is deterministic if $\forall \eta, \eta', \eta'' \in S$, $\forall sa \in$ SA : $\langle \eta, sa, \eta' \rangle \in T \wedge \langle \eta, sa, \eta'' \rangle \in T \Rightarrow \eta' = \eta''$.

Algorithms for symbolically executing symbolic transition systems have been proposed by Gaston et al. [53] and Jöbstl et al. [66]. However, as they do not deal with time, a new algorithm is presented in this thesis.

Algorithm 6.4 is an extended version of the algorithm proposed by Jöbstl et al. [66]. It requires two parameters: TIOSTS W is the input model to be symbolically executed and *ZSET* is the resulting zone-based symbolic execution tree. Firstly, a unique symbolic value is generated for each variable of V and each action parameter of P (Line 2). In Line 3, the

Algorithm 6.4: Symbolic Execution of $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$

```
1  symbolicExecution(TIOSTS $W$, ZSET $ZSET$){
2    $\varphi^0 \leftarrow$ map of variables of $V \cup P$ to symbolic values
3    $\eta^0 \leftarrow \langle l^0, \Theta, \varphi^0, Z^0 \rangle$
4    $addState(ZSET, \eta^0)$
5    $Unvisited \leftarrow \{\eta^0\}$
6    while $Unvisited \neq \emptyset$ do
7      pick and remove some $\eta = \langle l, \pi, \varphi, Z \rangle$ from Unvisited
8      for all $\langle l, a, G, A, y, l' \rangle \in \mathcal{T}$ do
9        $\mu_{sa} \leftarrow$ list of unique symbolic values for every parameter of $a$
10       $\varphi_{sa} \leftarrow$ map of action parameters to symbolic values
11       $sa \leftarrow \langle a, \mu_{sa}, \varphi_{sa} \rangle$
12       $\pi' \leftarrow \pi \wedge \varphi(\varphi_{sa}(G^D))$ // $G^D$ is the data guard of $G$
13       $\varphi' \leftarrow \varphi \circ \varphi_{sa} \circ A^D$ // $A^D$ represents data assignments of $A$
14       $Z' \leftarrow [A^C \leftarrow 0](G^C \cap \vec{Z})$ // $A^C$ is the set of clocks to reset and $G^C$ is the clock guard of $G$
15       $\eta' = \langle l', \pi', \varphi', Z' \rangle$
16       if ($isReachable(\eta') \wedge \neg(upperBoundReached(l')) \wedge \eta' \not\subseteq \eta'' \ \forall \eta'' \in ZSET$) then
17           $Unvisited \leftarrow Unvisited \cup \{\eta'\}$
18           $addState(ZSET, \eta')$
19           $addTransition(ZSET, \langle \eta, sa, \eta' \rangle)$
20       end if
21     end for
22   end while
23 }
```

first state η^0 of *ZSET* is defined considering the initial location of W, the initial condition of W as first path condition, the mapping defined in Line 2, and the initial clock zone (i.e., all clocks set to zero). Once defined, the first state η^0 is added to *ZSET* (Line 4).

The state η^0 is added to the set of states to be visited (Line 5). As long as there are unvisited states (Line 6), the algorithm picks and remove some state η from *Unvisited* (Line 7). The ZSES η refers to a location l of W and the loop in Line 8 processes all transitions from l.

The symbolic action sa is computed from the action a, attributing unique symbolic values for every parameter of a (Line 9) and mapping the original action parameter names to the defined symbolic values (Line 10).

Once the symbolic action has been defined (Line 11), the next step of Algorithm 6.4 is to compute the target state η'. Thus, the path condition π' for η' is defined (Line 12) as the conjunction of π with the guard G^D (i.e., the data guard of G) considering the mappings φ and φ_{sa}. The mapping φ' is defined through $\varphi \circ \varphi_{sa} \circ A^D$ (Line 13), where A^D represents data assignments of A and $\circ$ denotes function composition.

Z' is defined in Line 14. The successor of Z is defined by letting time elapse ($\vec{Z}$), taking the intersection with the clock guard G^C, and finally updating the values of clocks that are reset (i.e., clocks in A^C).

Once π', φ', and Z' have been defined, the target state η' is created in Line 15. Finally, η' is added to the set of states to be visited (Line 17) and a new transition labelled by sa connecting η to η' is added to *ZSET* (Lines 18 and 19), if the following conditions are satisfied (Line 16):

1. The state η' is reachable, that is, the path condition π' is satisfiable and the zone Z' is not empty;

2. The number of ZSESs in the current path that correspond to the location l' does not exceed a certain bound. This checking is needed to avoid infinite ZSETs in the case where there are loops in the specification whose number of iterations depends on values assigned to parameters and variables [66];

3. $\eta' \not\subseteq \eta'' \quad \forall \eta'' \in$ *ZSET* according to Definition 6.7, where the state inclusion of Gaston et al. [53] was extended to deal with zones.

Definition 6.7 (ZSESs Comparison). *Let $\eta = \langle l, \pi, \varphi, Z \rangle$ and $\eta' = \langle l', \pi', \varphi', Z' \rangle$ be two zone-based symbolic extended states. ZSES η' is included in ZSES η, that is, $\eta' \subseteq \eta$, if and only if:*

1. *$l' = l$;*

2. *$(\pi' \wedge \bigwedge_{x \in A^D}(x = \varphi'(x))) \Rightarrow (\pi \wedge \bigwedge_{x \in A^D}(x = \varphi(x)))$ is a tautology, where A^D represents data assignments of the TIOSTS;*

3. *$Z' \subseteq Z$.*

◇

As the implementation of the proposed algorithm for symbolic execution depends on tools related to concepts that are outside the scope of this thesis such as zones and constraint solving, the running time of the algorithm is described independently of implementation strategies and tools. Thus, the running time of Algorithm 6.4 is $O(\mid L \mid$ $+$ Cost(reachability analysis) $+$ Cost(ZSESs comparison)$)$, where $\mid L \mid$ is the number of locations of the TIOSTS W, Cost(reachability analysis) is the cost to verify whether a ZSES is reachable, and Cost(ZSESs comparison)$)$ is the cost to compare two ZSESs.

Figure 6.5 presents the ZSET obtained from the symbolic execution of the synchronous product shown in Figure 6.4.

6.2.4 Test Case Selection

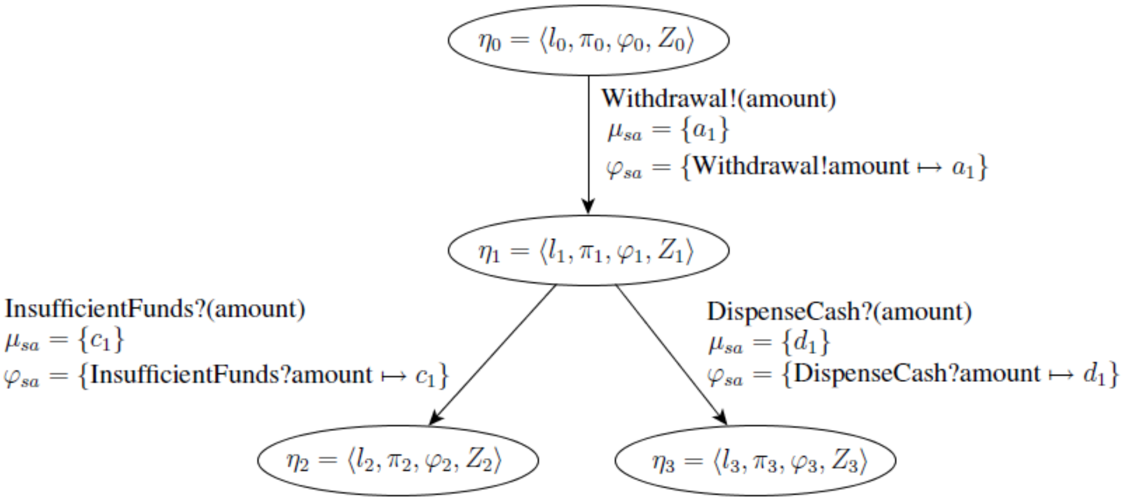
Once all possible traces have been identified by symbolic execution, the next step is to select a test case by choosing a trace that leads to an *Accept* state. For this, it is necessary to select a subtree of the generated ZSET called test tree. Finally, the selected test tree is translated into a test case (see Subsection 6.2.5).

The strategy used for the selection of the test tree is the same proposed by Jöbstl *et al.* [66], which is similar to the strategy of the TGV tool [62]. The idea is to select one reachable *Accept* state and perform a backward traversal to the root ZSES. Finally, a forward traversal is performed in order to extend the selected path to a test tree by adding missing inputs that are allowed by the specification. These missing inputs are possible outputs of the SUT and they are important to avoid fail verdicts on outputs allowed by the specification. In this case, the verdict is **Inconclusive**. Note that the forward traversal ensures the controllability of the generated test tree (i.e. test cases do not have the choice between inputs and outputs, or between several outputs).

The test tree from the ZSET in Figure 6.5 is the same ZSET since there is only one path leading to an *Accept* state and the addition of missing inputs leads to the whole ZSET.

6.2.5 Test Tree Transformation

Considering the conformance testing framework defined in Section 6.1, test cases are timed input-output symbolic transition systems. Thus, the last step of the test case generation


$\eta_0 = \langle l_0, \pi_0, \varphi_0, Z_0 \rangle$
Withdrawal!(amount)
$\mu_{sa} = \{a_1\}$
$\varphi_{sa} = \{\text{Withdrawal!amount} \mapsto a_1\}$
$\eta_1 = \langle l_1, \pi_1, \varphi_1, Z_1 \rangle$
InsufficientFunds?(amount)
$\mu_{sa} = \{c_1\}$
$\varphi_{sa} = \{\text{InsufficientFunds?amount} \mapsto c_1\}$
DispenseCash?(amount)
$\mu_{sa} = \{d_1\}$
$\varphi_{sa} = \{\text{DispenseCash?amount} \mapsto d_1\}$
$\eta_2 = \langle l_2, \pi_2, \varphi_2, Z_2 \rangle$
$\eta_3 = \langle l_3, \pi_3, \varphi_3, Z_3 \rangle$

$l_0 = \text{Idle\_EnterAmount}$
$\pi_0 = b_0 > 0$
$\varphi_0 = \{\text{balance} \mapsto b_0, \text{withdrawalValue} \mapsto w_0, \text{Withdrawal!amount} \mapsto a_0,$
$\text{InsufficientFunds?amount} \mapsto c_0, \text{DispenseCash?amount} \mapsto d_0\}$
$Z_0 = \text{clock} = 0$

$l_1 = \text{Verify\_Verify}$
$\pi_1 = b_0 > 0 \ \text{AND} \ a_1 > 0$
$\varphi_1 = \{\text{balance} \mapsto b_0, \text{withdrawalValue} \mapsto a_1, \text{Withdrawal!amount} \mapsto a_1,$
$\text{InsufficientFunds?amount} \mapsto c_0, \text{DispenseCash?amount} \mapsto d_0\}$
$Z_1 = \text{clock} = 0$

$l_2 = \text{Print\_Reject}$
$\pi_2 = b_0 > 0 \ \text{AND} \ a_1 > 0 \ \text{AND} \ c_1 = a_1 \ \text{AND} \ a_1 > b_0$
$\varphi_2 = \{\text{balance} \mapsto b_0, \text{withdrawalValue} \mapsto a_1, \text{Withdrawal!amount} \mapsto a_1,$
$\text{InsufficientFunds?amount} \mapsto c_1, \text{DispenseCash?amount} \mapsto d_0\}$
$Z_2 = \text{clock} \leq 2$

$l_3 = \text{Idle\_Accept}$
$\pi_3 = b_0 > 0 \ \text{AND} \ a_1 > 0 \ \text{AND} \ d_1 = a_1 \ \text{AND} \ a_1 \leq b_0$
$\varphi_3 = \{\text{balance} \mapsto b_0 - a_1, \text{withdrawalValue} \mapsto a_1, \text{Withdrawal!amount} \mapsto a_1,$
$\text{InsufficientFunds?amount} \mapsto c_0, \text{DispenseCash?amount} \mapsto d_1\}$
$Z_3 = \text{clock} \leq 10$

Figure 6.5: Zone-Based Symbolic Execution Tree of the TIOSTS of Figure 6.4

process (Figure 6.2) is to translate the selected test tree *TT* $= \langle S, \text{SA}, \eta^0, T \rangle$ into a test case TIOSTS *TC* $= \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$.

The test tree translation operation is described by Algorithm 6.5. It requires three parameters: TIOSTS *SP* is the synchronous product from which the test tree was obtained, ZSET *ZSET* is the test tree, and TIOSTS *TC* is the resulting test case.

The data of *TC* (i.e. $V \cup P$) is defined by symbolic values of *ZSET* (Lines 2 and 3). As in [66], the symbolic values are considered as variables and parameters of the test case. Let $\eta^0 = \langle l^0, \pi^0, \varphi^0, Z^0 \rangle$ be the initial state of *ZSET*, then the initial condition of *TC* is π^0

Algorithm 6.5: Test Tree Translation Algorithm

```
1  ZSET2TC(TIOSTS SP, ZSET ZSET, TIOSTS TC){
2     $V_{TC} \leftarrow$ symbolic values of variables of ZSET
3     $P_{TC} \leftarrow$ symbolic values of parameters of ZSET
4     // Let $\eta^0 = \langle l^0, \pi^0, \varphi^0, Z^0 \rangle$ be the initial state of ZSET
5     $\Theta_{TC} \leftarrow \pi^0$
6     $L_{TC} \leftarrow S$
7     $l^0_{TC} \leftarrow \eta^0$
8     $\Sigma_{TC} \leftarrow \bigcup_{\langle a, \mu_{sa}, \varphi_{sa} \rangle \in SA} a$
9     $C_{TC} \leftarrow C_{SP}$
10    for all $\langle \eta, sa, \eta' \rangle \in T_{ZSET}$ do
11       $l \leftarrow \eta$
12       $a \leftarrow$ action of $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$ with parameters of $\mu_{sa}$
13       $G \leftarrow$ conjunction of path condition of $\eta'$ with clock guards associated with $a$ in SP
14       $A \leftarrow$ clock resets associated with $a$ in SP
15       $y \leftarrow$ deadline associated with $a$ in SP
16       $l' \leftarrow \eta'$
17       addTransition(TC, $\langle l, a, G, A, y, l' \rangle$)
18    end for
19 }
```

(Line 5), the set of locations is S (Line 6), the initial location is η^0 (Line 7), the alphabet is $\bigcup_{\langle a, \mu_{sa}, \varphi_{sa} \rangle \in SA} a$ (Line 8), and the set of clocks is the same as the synchronous product *SP*, that is, C_{SP} (Line 9).

All transitions $\langle \eta, sa, \eta' \rangle \in T$ of *ZSET* are analysed in Lines from 10 to 18. Each transition of *ZSET* leads to the creation of a new transition $\langle l, a, G, A, y, l' \rangle \in \mathcal{T}$ in the test case. Thus, the source location is η, the action of the new transition is the action of $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$ with parameters of μ_{sa}, the conjunction of the path condition of η' with clock guards associated with a in *SP* is the guard, the assignments are defined based on clock resets associated with a in *SP*, the deadline is the same as the one associated with a in *SP*, and the target location is η'.

Using the asymptotic notation, the running time of Algorithm 6.5 is $O(|T|)$, where $|T|$ is the number of transitions of the *ZSET* test tree.

Figure 6.6 presents the test case obtained from the ZSET of Figure 6.5. It starts by performing a withdrawal transaction in the ATM system and resetting the clock to zero. Then it expects to receive the money. If the ATM system dispenses the expected money in at most 10 time units, the verdict is **Pass**, that is, the implementation is in conformance with the specification and the test purpose. If the ATM system indicates insufficient funds in at most 2 time units, the verdict is **Inconclusive** (i.e. the implementation conforms to the specification but the desired behaviour was not observed). Finally, if either an unspecified output is received or a time requirement is not satisfied, the verdict is **Fail**.

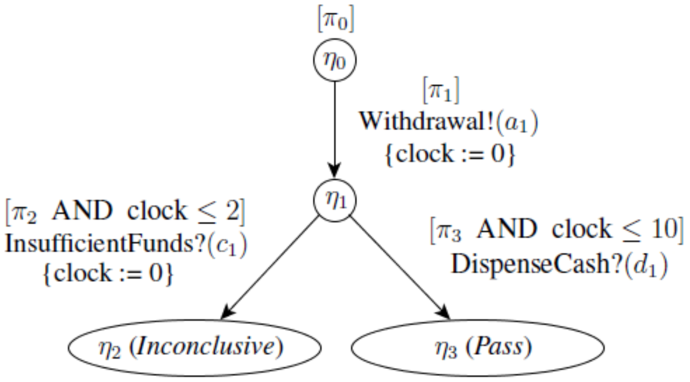


Figure 6.6: Test Case Obtained from the ZSET of Figure 6.5

6.3 Properties of the Test Cases

This section comments on properties of the test cases generated by the process presented in Section 6.2. The execution of test cases must be formalised in order to establish some properties such as soundness and exhaustiveness, where the conformance relation is linked to the verdicts.

The generated test cases are considered as a mechanism for guiding the execution of the implementation. Thus, conformance checking is performed in two steps, in an offline way. Firstly, the implementation is executed, guided by test cases, and all information needed to check conformance (e.g., input actions, responses, and time associated with responses) are logged into a file. Considering that the SUT runs on a real-time environment such as a real-time operating system, it is important that the implementation logs its own information in order to reduce the number of processes and consequently avoid introduction of noise in the results.

As said, each SUT execution produces a log describing the exercised scenario. This log is an *observable trace* (defined in Subsection 5.2), which is a specific sequence of observable discrete and time-elapsing actions. For example, considering the TIOSTS of Figure 5.1 an observable trace of a scenario where a withdrawal transaction is successfully done in 5 time units could be represented by $\sigma_{\text{SUT}} = 0\ \textit{Withdrawal}?(100)\ 5\ \textit{DispenseCash}!(100)$.

Let $[\![TC]\!] = \langle S, S^0, Act, T\rangle$ be the TIOLTS semantics of the test case $TC = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$. Thus, an observable trace of $\mathcal{I}$ can be checked with respect to the test case through the TIOLTS parallel composition defined by Krichen and Tripakis [78]. In this case, each trace $\sigma \in \textit{Traces}([\![TC]\!] \mid\mid \textit{ObservableTraces}(\mathcal{I}))$ is associated with one of the following scenarios:

- If all outputs of TC are executed and all inputs are observed on time, then the resulting verdict is **Pass**, that is, $\textit{verdict}(\sigma) = \textbf{Pass} \triangleq S^0\ \textit{after}\ \sigma \subseteq \textit{Pass}$;
- If, at any moment, any unspecified input is observed by the test case or some time requirement is not met, the conformance checking is stopped and the resulting verdict is **Fail**, that is, $\textit{verdict}(\sigma) = \textbf{Fail} \triangleq S^0\ \textit{after}\ \sigma \subseteq \textit{Fail}$;
- We denote $\textit{verdict}(\sigma) = \textbf{Inconclusive} \triangleq S^0\ \textit{after}\ \sigma \subseteq \textit{Inconclusive}$ for two situations: if $\mathcal{I}$, at any moment, blocks or spends a lot of time to emit an output; and if the outputs of $\mathcal{I}$ are specified by $\mathcal{S}$ but the behaviour specified by a test purpose is not exhibited.

Given the possible situations with their respective verdicts, the rejection of $\mathcal{I}$ by a test case TC is formally defined as follows:

Definition 6.8 (may reject)**.** TC may reject $\mathcal{I} \triangleq \exists \sigma \in \text{Traces}([\![TC]\!] \mid\mid \text{ObservableTraces}(\mathcal{I})) : \text{verdict}(\sigma) = \textbf{\textit{Fail}}$. ◇

The conformance relation of an implementation with its specification is decided based on verdicts obtained with the execution of test cases. So, Definition 6.9 formally relates the **tioco** relation to the verdicts considering some properties of test cases and test suites.

Definition 6.9 (Soundness and Exhaustiveness)**.** *A test case TC is* sound *for $\mathcal{S}$ and* ***tioco*** *if $\forall \mathcal{I}$, $\mathcal{I}$* ***tioco*** *$\mathcal{S} \Rightarrow \neg($$TC$ may reject $\mathcal{I}$*). A test suite is* sound *if all its test cases are* sound *and it is* exhaustive *for $\mathcal{S}$ and* ***tioco*** *if $\forall \mathcal{I}$, $\neg(\mathcal{I}$* ***tioco*** *$\mathcal{S}) \Rightarrow \exists TC : TC$* may reject $\mathcal{I}$*. Finally, a test suite is* complete *if it is both* sound *and* exhaustive. ◇

Informally, a test suite is sound if correct implementations are never rejected. On the other hand, a test suite is exhaustive if all non-conforming implementations are rejected. A test suite that can identify all conforming and non-conforming implementations is called complete. Since a complete test suite is a very strong requirement for practical testing, sound test suites are more commonly accepted. In this context, the test cases generated by our approach have the properties stated in Theorem 6.1.

Theorem 6.1. *For every specification $\mathcal{S}$, all test suites generated by our approach are sound. Moreover, test suites can be considered as being exhaustive when they refer to specific scenarios defined by test purposes.* ◇

The proof of Theorem 6.1 is not detailed here but the main ideas are discussed (see detailed proofs in Appendix A). For soundness, we need to prove that if a test case TC may reject $\mathcal{I}$ (implementing the specification $\mathcal{S}$), then $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$. In this case, we only need to prove that a **Fail** verdict only occurs if $\mathcal{I}$ emits an unspecified output or some time requirement is not met. In our approach, test cases are generated based on symbolic execution of specifications. This approach allows to identify all possible traces of a specification. Thus, the unique case where a **Fail** verdict occurs is exactly when $\mathcal{I}$ emits an unexpected output or some time requirements is not satisfied. For exhaustiveness, we need to prove that for every non-conforming $\mathcal{I}$ there is a test purpose TP and a way of generating a test case TC from $\mathcal{S}$ and TP, such that TC may reject $\mathcal{I}$. Given that $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$, then there is a trace σ of $\mathcal{S}$ such that an output of $\mathcal{I}$ after σ is not allowed by $\mathcal{S}$. In this case, a TP can be defined based on σ and used to generate test cases where $\mathcal{I}$ may be rejected.

6.4 Concluding Remarks

This chapter presented an approach to conformance testing of real-time systems based on the use of a symbolic model that abstracts both time and data in order to broadening the application of conformance testing in this field. It also described the test case generation process and discussed some properties of the generated test cases.

The presented test case generation process is completely automated by a tool developed to support the proposed approach. In order to check the satisfiability of path conditions and verify state inclusion w.r.t. data we are using the CVC3 SMT Solver[2]. As the satisfiability of data guards is assumed to be decidable, the CVC3 SMT Solver arises as a promising tool [111]. Moreover, data guards of a TIOSTS are expressed using the same notation as the notation used by the CVC3 SMT Solver, which facilitates the use of the tool and does not require any translation. However, it is important to note that our approach is limited to the types supported by this solver such as Boolean, integer, real, arrays, records, etc.

All operations related to zones used in Algorithm 6.4 are provided by UPPAAL DBM Library[3]. The same library is also used to verify the state inclusion w.r.t. zones.

[2] http://www.cs.nyu.edu/acsys/cvc3

[3] http://www.cs.aau.dk/˜adavid/UDBM

Chapter 7

Interruption Testing of Real-Time Systems

This chapter describes how to model and test interruptions using the TIOSTS-based conformance testing framework presented in Chapters 5 and 6. The defined interruption testing strategy is only a way of modelling using TIOSTS models and no modification in the theory is needed. This strategy is the same as the one presented in [7], where interruptions are modelled by non-real-time symbolic models. Furthermore, as for the ALTS models presented in Chapter 3, the TIOSTS-based interruption testing strategy makes it possible to combine interruptions at different points and allows to select test cases based on test purposes.

7.1 Modelling and Testing Interruptions in Real-Time Systems

The TIOSTS model proposed in Chapter 5 can be used to model interruptions. The idea is to take advantage of the use of variables and action parameters in order to guarantee that once the main flow has been interrupted, it can continue its execution from the same point where the interruption had started.

In order to model interruptions, consider the existence of two models: one TIOSTS representing the main flow (Figure 7.1, locations from 0 to 20), that is, the application that can be interrupted; and another TIOSTS representing the interruption (Figure 7.1, locations from 21 to 28). Thus, the main flow can be linked with the interruption model through the following steps:

1. Identify the point (location) where the interruption can occur (Figure 7.1, location 10);
2. Link this point to the interruption behaviour using a transition labelled as follows: the guard is *intCode = X and choice = 0*, where *X* is an integer that uniquely identifies this point of interruption; the action is *Interrupt?(intCode)*; and the assignment is *choice := intCode* (Figure 7.1, transition from location 10 to 21). Notice that the value of the parameter *intCode* is saved into the *choice* variable.
3. Connect the last action of the interruption behaviour to the same point where the interruption started using a transition labelled with the guard *choice = X*, where *X* is the same value that uniquely identifies this point of interruption (Figure 7.1, transition from location 28 to 10). This guard is used to guarantee that the main flow continues its execution from the same point where it had been interrupted. For instance, if an interruption begins with the parameter *intCode* equals to 1, then it must finish performing the action that has the following guard: *choice = 1*.

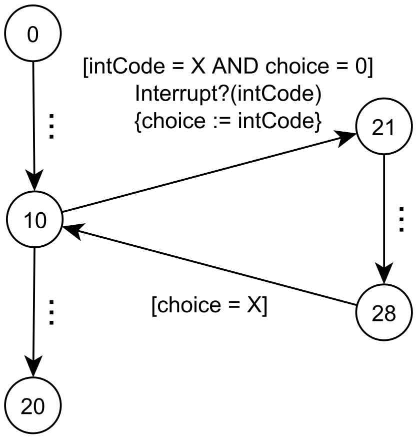


Figure 7.1: Modelling an Abstract Interruption

The test case generation strategy where only one interruption is allowed for each test case is achieved because of the second part of the guard (*choice = 0*) associated to the *Interrupt* action. When an interruption is allowed, the value of the *choice* variable is changed to any value different from zero, then all other interruptions are automatically discarded during the test case generation.

The defined steps must be performed for all points where interruptions can occur. As a complete model with all possibilities of interruption represents many scenarios, the test selection strategy based on test purposes defined in Chapter 6 is used for testing specific interruptions in specific scenarios. For this, it is enough to use the *Interrupt* action in the test purpose carrying the integer that identifies the selected point. For generating test cases without interruptions, the *Interrupt* action is taken to the *Reject* location.

In order to test interruptions using TIOSTS models, the same test architecture presented in Chapter 3 is adopted. Thus, the environment is assumed to be fully controllable by the tester.

7.2 Instantiating the Strategy with an Example

A real-time version of the mobile phone application described in Chapter 3 is used for describing how to deal with interruptions using the TIOSTS formalism. Now, the action of removing a message from inbox after selecting the "Remove" option must be performed in at most 2000 milliseconds. Also, only unblocked messages can be removed and the main application can be interrupted at some points by the Incoming Alert interruption. As said in Chapter 3, this interruption specifies the arrival of a simple text message displayed inside a dialog box. Figure 7.2 shows the TIOSTS model that represents the described behaviour, where locations from 1 to 12 represent the behaviour of removing a message from inbox and locations from 13 to 16 represent the occurrence of interruptions.

As we can see, in Figure 7.2, the interruption model is connected to the feature that can be interrupted (the main flow) using the *Interrupt* action carrying a parameter (*intCode*) that identifies the place where the interruption is allowed. Then, the *intCode* parameter is saved into the *choice* variable. Each point where an interruption is allowed has a different integer value associated with it. Another important information is in the last action of the interruption, where there is a guard used to guarantee the return to the correct point of the main flow.

As discussed in Chapter 3, an interruption can occur at infinite points during the system execution, but in the tester's point of view, an interruption can only be observed after an output of the SUT. Thus, the TIOSTS model of Figure 7.2 represents all possibilities of

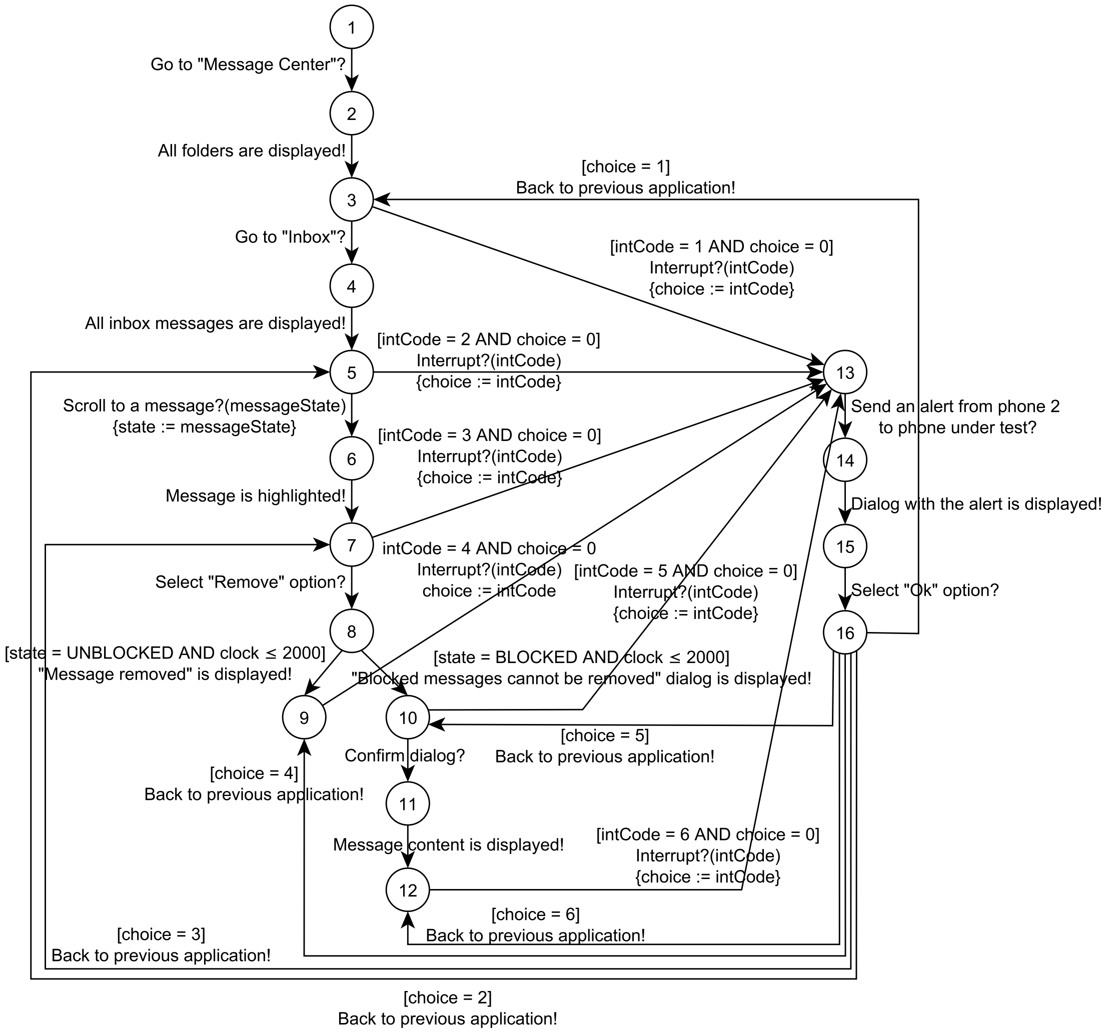
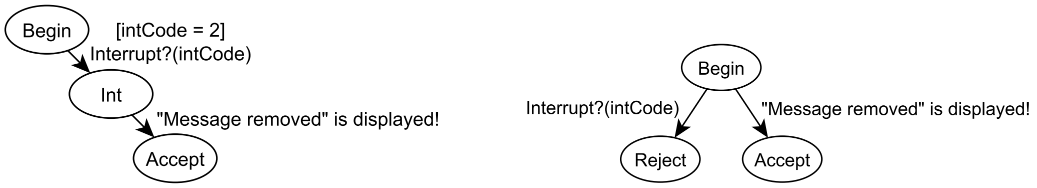

1
Go to "Message Center"?
2
All folders are displayed!
[choice = 1]
Back to previous application!
3
Go to "Inbox"?
[intCode = 1 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
4
All inbox messages are displayed!
[intCode = 2 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
5
13
Scroll to a message?(messageState)
{state := messageState}
Send an alert from phone 2
to phone under test?
[intCode = 3 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
6
14
Message is highlighted!
Dialog with the alert is displayed!
7
intCode = 4 AND choice = 0
Interrupt?(intCode)
choice := intCode
15
[intCode = 5 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
Select "Remove" option?
Select "Ok" option?
8
16
[state = UNBLOCKED AND clock ≤ 2000]
"Message removed" is displayed!
[state = BLOCKED AND clock ≤ 2000]
"Blocked messages cannot be removed" dialog is displayed!
9
10
[choice = 5]
Back to previous application!
Confirm dialog?
[choice = 4]
Back to previous application!
11
Message content is displayed!
[intCode = 6 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
12
[choice = 6]
Back to previous application!
[choice = 3]
Back to previous application!
[choice = 2]
Back to previous application!

Figure 7.2: Real-Time Version of the Remove Message Behaviour with Interruptions

interruption from the tester's point of view.

Once the system is specified using the TIOSTS formalism, the next step is to define test purposes in order to check specific interruptions at some points. Considering the specification in Figure 7.2, a test purpose can be defined in order to verify the scenario where an alert appears when the user is accessing the inbox folder. As the selected interruption point corresponds to the second output of the specification, the action *Interrupt* must carry the integer 2. Next, the last action of the selected behaviour (the scenario where the message is removed) is appended to the test purpose. Figure 7.3(a) presents the test purpose defined above and Figure 7.4 shows the obtained test case. Note that in the generated test case (Figure 7.4) the interruption occurs only in one specific point.

Another test purpose can be defined in order to test a scenario where a message is re-



(a) Scenario with Interruption

(b) Scenario without Interruption

Figure 7.3: Test Purposes

moved and all interruptions are not allowed. This test purpose is presented in Figure 7.3(b). Note that to prohibit all interruptions it is enough to take the *Interrupt* action to the *Reject* location. The other action of the test purpose (*"Message removed" is displayed!*) is used to select the scenario where the message is removed. The generated test case is shown in Figure 7.5.

7.3 Concluding Remarks

This chapter presented a strategy developed for modelling and testing interruptions that is based on the symbolic model proposed in Chapter 5. The presented strategy makes it possible for interruptions to be combined at different points of possibly different flows of execution. Moreover, test purposes can be used to select specific interruptions to be tested. Finally, it is important to remark that this proposed strategy is only a specific way of modelling using the TIOSTS notation. No modifications in the theory and algorithms are needed.

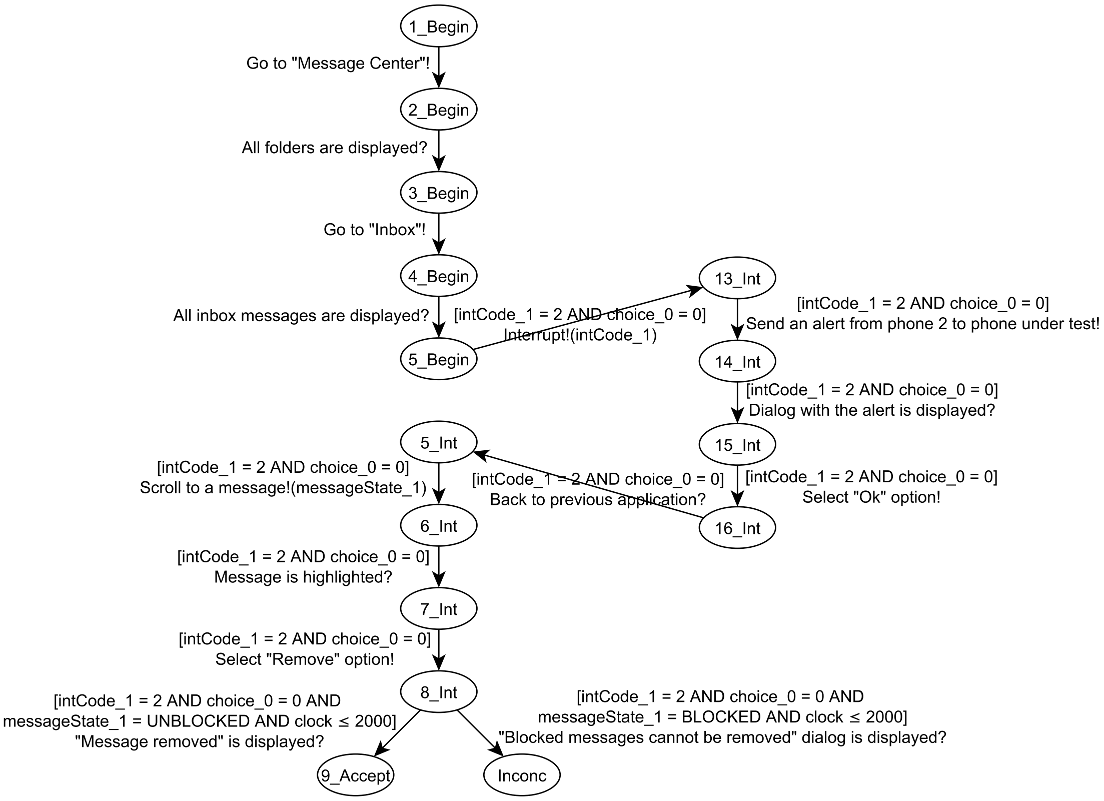

1_Begin
Go to "Message Center"!
2_Begin
All folders are displayed?
3_Begin
Go to "Inbox"!
4_Begin
All inbox messages are displayed?
5_Begin
[intCode_1 = 2 AND choice_0 = 0]
Interrupt!(intCode_1)
13_Int
[intCode_1 = 2 AND choice_0 = 0]
Send an alert from phone 2 to phone under test!
14_Int
[intCode_1 = 2 AND choice_0 = 0]
Dialog with the alert is displayed?
15_Int
[intCode_1 = 2 AND choice_0 = 0]
Select "Ok" option!
16_Int
[intCode_1 = 2 AND choice_0 = 0]
Back to previous application?
5_Int
[intCode_1 = 2 AND choice_0 = 0]
Scroll to a message!(messageState_1)
6_Int
[intCode_1 = 2 AND choice_0 = 0]
Message is highlighted?
7_Int
[intCode_1 = 2 AND choice_0 = 0]
Select "Remove" option!
8_Int
[intCode_1 = 2 AND choice_0 = 0 AND
messageState_1 = UNBLOCKED AND clock ≤ 2000]
"Message removed" is displayed?
[intCode_1 = 2 AND choice_0 = 0 AND
messageState_1 = BLOCKED AND clock ≤ 2000]
"Blocked messages cannot be removed" dialog is displayed?
9_Accept
Inconc

Figure 7.4: Test Case with Interruption

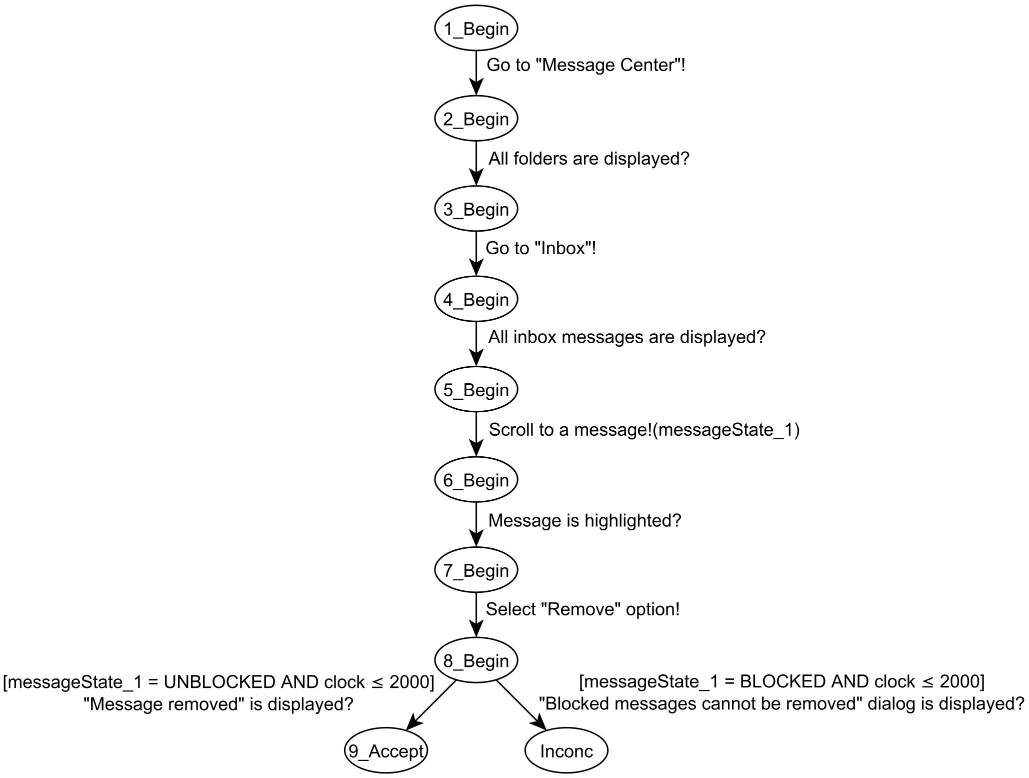

1_Begin
Go to "Message Center"!
2_Begin
All folders are displayed?
3_Begin
Go to "Inbox"!
4_Begin
All inbox messages are displayed?
5_Begin
Scroll to a message!(messageState_1)
6_Begin
Message is highlighted?
7_Begin
Select "Remove" option!
8_Begin
[messageState_1 = UNBLOCKED AND clock ≤ 2000]
"Message removed" is displayed?
[messageState_1 = BLOCKED AND clock ≤ 2000]
"Blocked messages cannot be removed" dialog is displayed?
9_Accept
Inconc

Figure 7.5: Test Case without Interruption

Chapter 8

Case Studies

The objective of this chapter is to present some case studies performed in order to evaluate the practical application of the symbolic model-based testing approach proposed in this thesis. The application of the developed approach is evaluated using the burglar alarm system and the automatic guided vehicle case studies, respectively.

8.1 The Burglar Alarm System

The first case study is aimed at generating and executing test cases for the burglar alarm system described in Section 4.3. For this, a simplified implementation was developed to run on a real-time operating system named FreeRTOS [112], a mini-kernel that can be used to develop real-time systems for embedded devices. The alarm system case study is useful because it is possible to execute test cases and it allows us to show how scenarios with interruptions can be checked.

The main objective of this case study is to assess the performance of the symbolic model-based approach developed for testing real-time systems. In order to achieve this objective we use the Goal/Question/Metric (GQM) paradigm [14], a mechanism for defining and evaluating goals using measurement.

8.1.1 The GQM Measurement Model

The GQM paradigm is a top-down systematic approach to evaluating goals based on an operational level. Thus, the first step is to define the goal to be evaluated (conceptual level). Secondly, at the operational level, questions are defined in order to characterize the measurement object with respect to desired quality criteria. Finally, a set of data is defined to answer each question in a quantitative level.

Figure 8.1 presents the GQM measurement model defined for this case study. The main goal is to evaluate the performance of the symbolic model-based approach developed for testing real-time systems. Thus, three questions were defined to characterize the measurement objects:

What is the effort required to use this approach? This question is intended to evaluate the effort required to apply all the test process from the building of the model to test case execution. For answering this question the following metric was defined: $E = E_1 + E_2 + E_3 + E_4 + E_5$, where:

- E_1 is the time spent to build the model using the TIOSTS formalism;
- E_2 is the time spent to define test purposes in order to test specific scenarios. Test purposes are also defined using the TIOSTS formalism;
- E_3 is the time spent to generate test cases using the prototype tool implementing the presented algorithms;
- E_4 is the time spent to implement the automatically generated test cases;
- E_5 is the time spent to execute the implemented test cases and evaluate the obtained results for emitting verdicts.

How effective is this approach? This question is intended to evaluate the effectiveness of the proposed approach w.r.t. fault coverage. The C metric, used to answer this question, indicates the ability of generated test cases of uncovering faults described by a previously defined fault model.

What percentage of test cases cannot be executed? This question is intended to identify the percentage of invalid test cases, represented by the metric I.

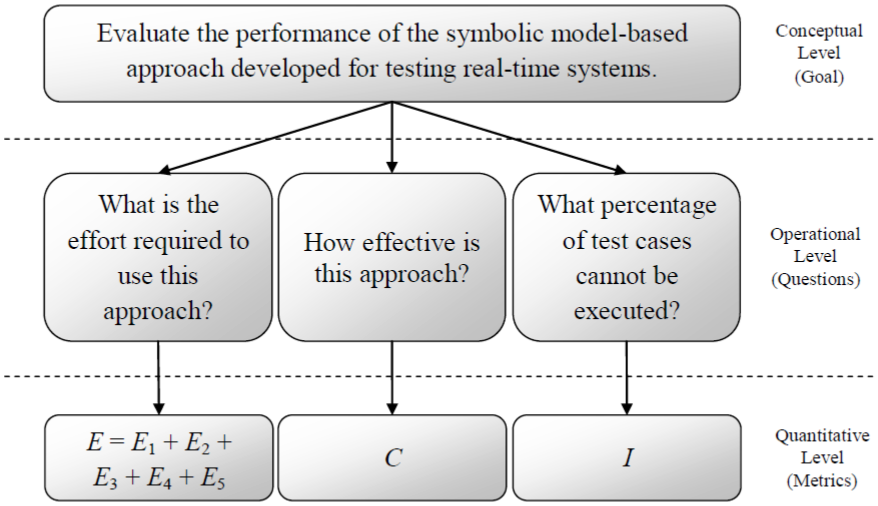

Evaluate the performance of the symbolic model-based approach developed for testing real-time systems.
Conceptual Level (Goal)
What is the effort required to use this approach?
How effective is this approach?
What percentage of test cases cannot be executed?
Operational Level (Questions)
$E = E_1 + E_2 + E_3 + E_4 + E_5$
C
I
Quantitative Level (Metrics)

Figure 8.1: Measurement Model for the Alarm System Case Study

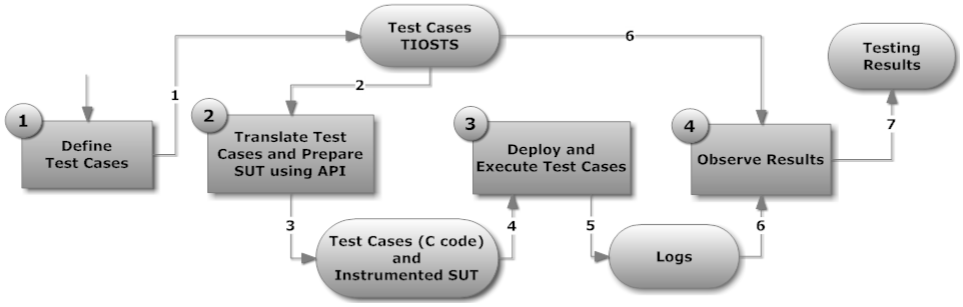

Test Cases TIOSTS
Testing Results
1 Define Test Cases
2 Translate Test Cases and Prepare SUT using API
3 Deploy and Execute Test Cases
4 Observe Results
Test Cases (C code) and Instrumented SUT
Logs
1
2
3
4
5
6
6
7

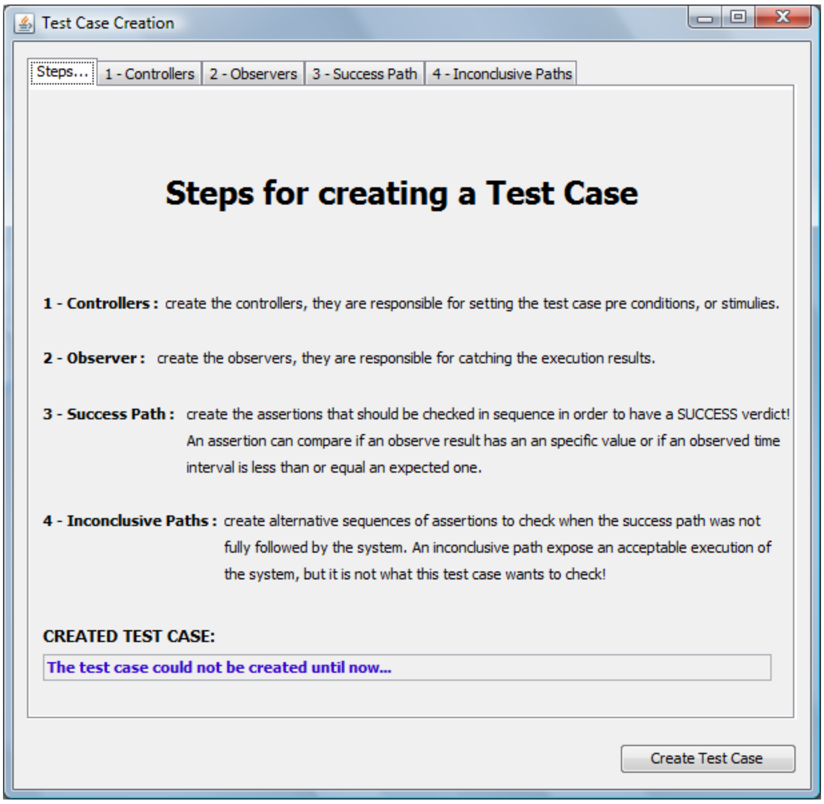
Figure 8.2: Testing Process

8.1.2 Case Study Definition

An initial infrastructure to support test execution in an actual real-time environment is investigated in [10]. However, the work presented in [90] extends the previous work by presenting an effective solution to support automation of test case execution for real-time systems.

The testing process considered in this case study is illustrated in Figure 8.2. It is divided into four well defined steps.

The first step is to define test cases according to the approach and theory presented in Chapters 5, 6, and 7. As there is a prototype tool implementing the proposed test case generation and selection strategy, the tester needs to manually instantiate a Java class to define the TIOSTS of the specification and the TIOSTS representing the test purpose. Once the specification and test purpose have been defined, the test case generation and selection is automatically performed.



Steps for creating a Test Case

1 - Controllers : create the controllers, they are responsible for setting the test case pre conditions, or stimulies.

2 - Observer : create the observers, they are responsible for catching the execution results.

3 - Success Path : create the assertions that should be checked in sequence in order to have a SUCCESS verdict! An assertion can compare if an observe result has an an specific value or if an observed time interval is less than or equal an expected one.

4 - Inconclusive Paths : create alternative sequences of assertions to check when the success path was not fully followed by the system. An inconclusive path expose an acceptable execution of the system, but it is not what this test case wants to check!

CREATED TEST CASE:

The test case could not be created until now...

Figure 8.3: Test Case Builder Application

At the second step, each test case is translated into C code. Another Java application has been implemented to support this activity as shown in Figure 8.3. Currently, the tester manually indicate all inputs and outputs of the generated test case in an interactive way and a C code implementing the test case is automatically generated. Furthermore, to make the system testable, a C API (see [90]) has been developed to instrument the code of the SUT. The SUT is instrumented for including Points of Control and Observation (PCOs). As the focus of this work is on functional testing, some examples of PCOs are the possibilities of observing values returned by functions, received messages, and timing associated with system responses. The instrumentation activity is manually performed by the tester using the implemented API.

The third step consists in the execution of the instrumented SUT guided by test cases. For this, a logging mechanism has been implemented in order to store all information needed to check the testing results. Considering that the SUT runs on a real-time environment such as a real-time operating system, it is important that the implementation logs its own information in order to reduce the number of processes and consequently avoid introduction of noise in the results. As we are dealing with RTES, each addition of code has a direct effect in the execution time of the application. Thus, test case execution can interfere with the flow of execution of the SUT, adding delays that can lead to false positives of failures. For minimizing this interference, all generated information is kept in the main memory. After the SUT execution, a simple text file is generated with the results. Logging frameworks such as log4c[1] are not suitable in this case since the quantity of dependencies forbid their execution within a dedicated hardware and the added code can cause a large delay at the actual execution time of the SUT.

The fourth step receives as input the text file with the execution results and provide verdicts for the tester. This step is not executed inside the execution platform level, but at the development platform level. An extension of the CUnit[2] framework has been developed for evaluating the text file with execution results and emit a verdict according to the test cases defined in the first step.

The case study was conducted by only one tester with large experience in TIOSTS models. Moreover, the case study aimed at testing only one scenario due to difficulties to completely execute the testing process. The scenario to be tested was based on a fault model profile based on common faults related to interruption testing [9] and potential faults in an implementation of a TIOSTS. The fault model given to the tester is specified in natural language and its description is defined as follows:

- After an interruption, the interrupted application does not maintain data previously received as input;
- After an interruption, the interrupted application does not continue its execution at the same point where it was interrupted;
- Unexpected outputs, when an implementation responds with an output not described in its specification;
- Clock guard restriction, when an implementation reduces an execution time range associated with an action;

[1] http://log4c.sourceforge.net

[2] http://cunit.sourceforge.net

- Clock guard widening, when an implementation increases an execution time range associated with an action.

It is important to remark that all information used as input for the execution of this case study was the high level description of the SUT presented in Section 4.3, the defined fault model, and the implementation of the SUT.

8.1.3 Case Study Results

This subsection presents and discusses the obtained results. The most critical scenario of the alarm system is the power failure exactly after some alarm has been triggered. In this case, the system must switch to backup power and continues its execution with the calling to the police and turning on the lights of the room where the sensor detected an intruder.

Considering the first activity of the testing process (see Figure 8.2), a TIOSTS specification was built along with a TIOSTS test purpose to check the defined scenario. Figures 8.4 and 8.5 present the defined specification and test purpose, respectively. However, all TIOSTS models automatically generated by the prototype tool are presented in Appendix B.

Table 8.1 summarizes the metrics collected during the execution of all activities of the test process. It is important to remark that the test case generation and implementation is performed in a different platform from the execution platform of the SUT. The development platform has the following characteristics: 2x3.00GHz CPU, 1536MB RAM, Ubuntu 10.10, CVC3 2.2, and UPPAAL DBM Library 2.0.7. The execution platform is based on FreeRTOS environment using the industrial PC (x86) port.

Considering the first activity of the testing process depicted in Figure 8.2, the tester spent 40 minutes to manually implement the TIOSTS representing the specification and 4 minutes to implement the TIOSTS test purpose (Table 8.1, lines 1 and 2, respectively). Once the specification and test purpose are built, the test case generation and selection is automatically performed in 3 seconds (Table 8.1, line 3).

The prototype tool generated 3 test cases (Table 8.1, line 4). Observing the specification presented in Figure 8.4 it is possible to realize that only considering the interruption after triggering an alarm, there are three possibilities. The first path is the scenario where a window breaking alarm is triggered (we denote TC1). At the second scenario, a door opening

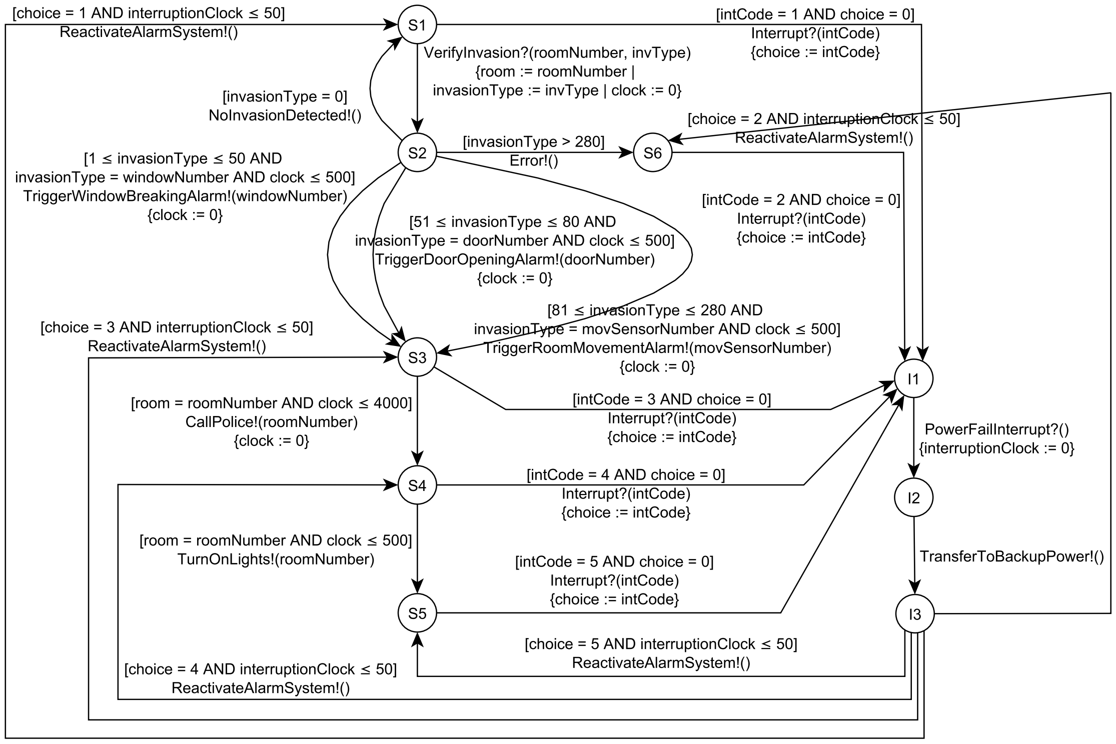

[choice = 1 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
S1
[intCode = 1 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
VerifyInvasion?(roomNumber, invType)
{room := roomNumber |
invasionType := invType | clock := 0}
[invasionType = 0]
NoInvasionDetected!()
[choice = 2 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
[1 ≤ invasionType ≤ 50 AND
invasionType = windowNumber AND clock ≤ 500]
TriggerWindowBreakingAlarm!(windowNumber)
{clock := 0}
S2
[invasionType > 280]
Error!()
S6
[intCode = 2 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
[51 ≤ invasionType ≤ 80 AND
invasionType = doorNumber AND clock ≤ 500]
TriggerDoorOpeningAlarm!(doorNumber)
{clock := 0}
[81 ≤ invasionType ≤ 280 AND
invasionType = movSensorNumber AND clock ≤ 500]
TriggerRoomMovementAlarm!(movSensorNumber)
{clock := 0}
[choice = 3 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
S3
I1
[room = roomNumber AND clock ≤ 4000]
CallPolice!(roomNumber)
{clock := 0}
[intCode = 3 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
PowerFailInterrupt?()
{interruptionClock := 0}
S4
[intCode = 4 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
I2
[room = roomNumber AND clock ≤ 500]
TurnOnLights!(roomNumber)
[intCode = 5 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
TransferToBackupPower!()
S5
I3
[choice = 5 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()
[choice = 4 AND interruptionClock ≤ 50]
ReactivateAlarmSystem!()

Figure 8.4: TIOSTS Specification for the Burglar Alarm System Case Study

alarm is triggered (TC2). The last scenario is the case where a room movement alarm is triggered (TC3).

After the generation of test cases, the testing process is completely executed for each of them from the second step. Considering only the TC1, the next activity is to implement it and prepare the environment to the execution. For this, three activities were performed:

1. The test case builder application (Figure 8.3) was used to generate the C code implementing the test case in 25 minutes (Table 8.1, line 5). For this, the tester manually indicated all inputs, outputs, and time requirements of TC1. It is important to mention that all defined information (inputs, outputs, and so on) can be reused in the definition of other test cases with the same information and the CVC3 SMT Solver is used for automatically defining the inputs of the test case.

2. An interface was implemented to allow the communication between the SUT and test driver. This activity was performed in 18 minutes (Table 8.1, line 6) and it is used for all test cases.

3. Finally, the SUT was instrumented for the execution of TC1. 6 minutes were spent to

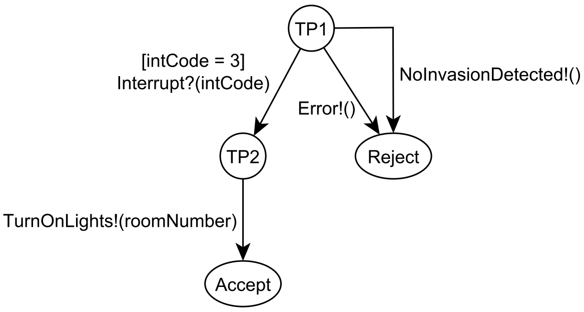
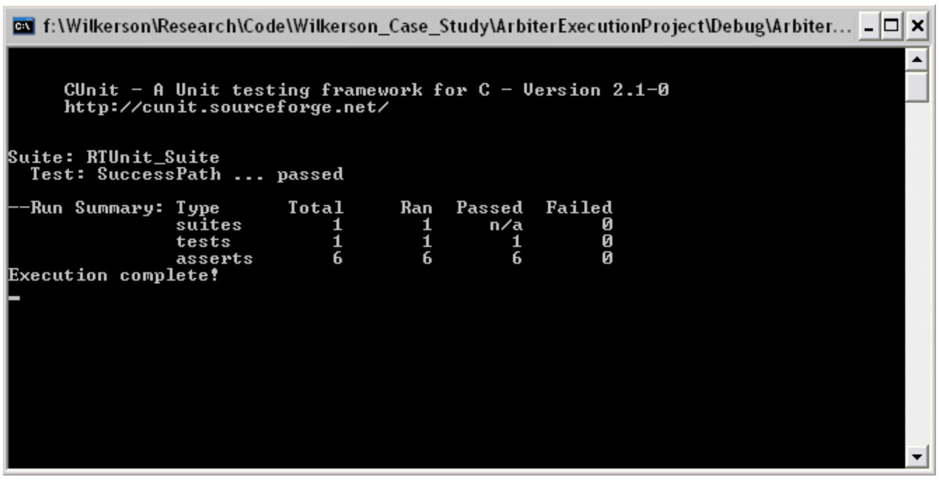


Figure 8.5: TIOSTS Test Purpose for the Burglar Alarm System Case Study


f:\Wilkerson\Research\Code\Wilkerson_Case_Study\ArbiterExecutionProject\Debug\Arbiter...

```
     CUnit - A Unit testing framework for C - Version 2.1-0
     http://cunit.sourceforge.net/


Suite: RTUnit_Suite
  Test: SuccessPath ... passed

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        1       1     n/a       0
               tests         1       1       1       0
               asserts       6       6       6       0
Execution complete!
```

Figure 8.6: Results of the TC1 Execution

perform this activity (Table 8.1, line 7).

The next step is to deploy the instrumented SUT, execute it to extract the generated log file, and evaluate the results. As this evaluation is automatically performed (see Figure 8.6) it takes little time (Table 8.1, line 8).

Considering the execution of TC2 (Table 8.1, lines from 9 to 11) and TC3 (Table 8.1, lines from 12 to 14), the testing process activities take less time than the activities related to the execution of TC1 because many information can be reused.

Since all needed information was collected, the questions of the defined measurement model can be answered. For the first question, "what is the effort required to use this approach?", we have the following effort required to test the defined scenario considering all

Table 8.1: Metrics of the Burglar Alarm System Case Study

	Metrics	**Time**
1	Definition and implementation of the TIOSTS specification	40 *min*
2	Definition and implementation of the TIOSTS test purpose	4 *min*
3	Test case generation time	3 *sec*
4	Number of Test Cases	3
5	Implementation of TC1	25 *min*
6	Implementation of an interface to allow the communication between SUT and test driver	18 *min*
7	Instrumentation of SUT for executing TC1	6 *min*
8	Evaluation of results for TC1	1 *sec*
9	Implementation of TC2	5 *min*
10	Instrumentation of SUT for executing TC2	5 *min*
11	Evaluation of results for TC2	1 *sec*
12	Implementation of TC3	5 *min*
13	Instrumentation of SUT for executing TC3	5 *min*
14	Evaluation of results for TC3	1 *sec*
15	Execution time of all test process for all generated test cases	113.1 *min*
16	Fault model coverage	100%
17	Number of invalid TCs	0 (0%)

generated test cases:

$$\begin{aligned} E &= E_1 + E_2 + E_3 + E_4 + E_5 \\ &= 40\ min + 4\ min + 3\ sec + 69\ min + 3\ sec \\ &= 113.1\ min \end{aligned}$$

The fault model profile defined in Subsection 8.1.2 was instantiated in order to evaluate the effectiveness of the approach proposed in this thesis. Thus, the following real defects were inserted in the SUT:

- After the power failure interruption, the calling to the police informs a wrong room number;

- After the power failure interruption, the SUT turns on the lights of the room where the sensor detected an intruder instead of calling to the police;
- After calling to the police, the SUT performs an unspecified output;
- The SUT performs the action of turning on the lights in more than 50 ms.

As the specification does not specify lower bounds as time requirements, it not possible to insert faults related to clock guard restrictions. As shown in Table 8.1, line 16, 100% of the defined fault model instance is covered by the defined test cases.

Finally, as all generated test cases were executed, no invalid test cases could be detected (Table 8.1, line 17).

8.2 The Automatic Guided Vehicle System

The automatic guided vehicle (AGV) system consists of a robot for autonomous navigation that is able to plan and execute predefined tasks. Automatic guided vehicles are used, for example, to transport materials in industries, for inspection in risk areas or deposits of toxic materials, etc.

Basically, the first step of an AGV system is to define the plan to be followed. After that, the execution is started. Sensors are used to drive the vehicle and allow it to overcome obstacles of the path. Deviations from the original path may be required because of obstacles and avoid collisions.

This case study is aimed at generating test cases for the AGV system. Test execution is not considered because we do not have an implementation of this system. Even thus, the AGV case study allows us to evaluate the test case generation activity including the generation of test cases for checking interruptions.

8.2.1 The GQM Measurement Model

Since test execution is not considered in this case study, another measurement model was defined (Figure 8.7). The goal is the same as in Section 8.1, but only two questions were defined to characterize the measurement objects:

What is the effort required to use this approach? This question is intended to evaluate the effort required to apply all the test process from the building of the model to test case generation. For answering this question the following metric was defined: $E = E_1 + E_2 + E_3$, where:

- E_1 is the time spent to build the model using the TIOSTS formalism;
- E_2 is the time spent to define test purposes in order to test specific scenarios.
- E_3 is the time spent to generate test cases using the developed prototype tool.

How effective is this approach? The objective of this question is to evaluate the effectiveness of the proposed approach w.r.t. fault coverage. Thus, the C metric indicates the ability of generated test cases for uncovering faults described by a previously defined fault model.

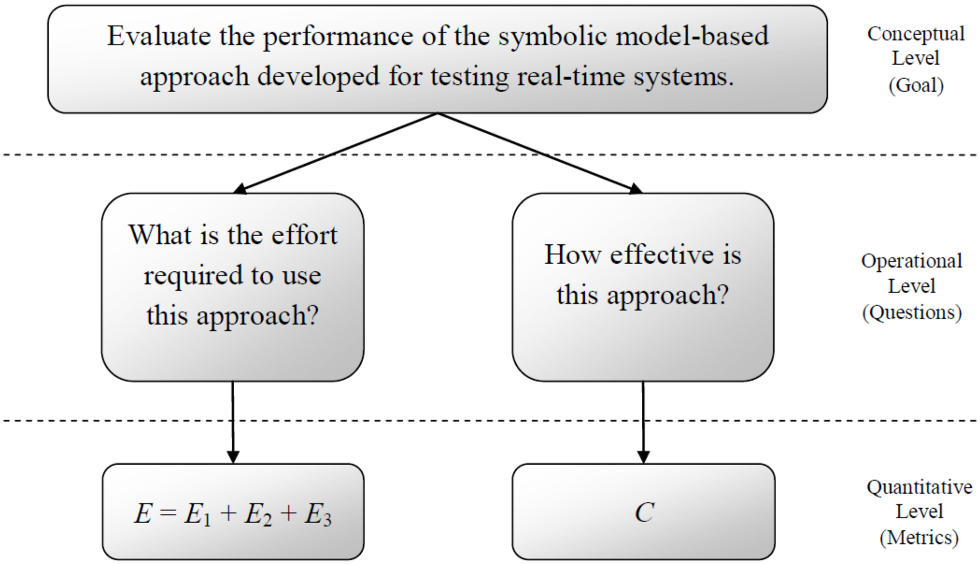

Evaluate the performance of the symbolic model-based approach developed for testing real-time systems.
Conceptual Level (Goal)
What is the effort required to use this approach?
How effective is this approach?
Operational Level (Questions)
$E = E_1 + E_2 + E_3$
C
Quantitative Level (Metrics)

Figure 8.7: Measurement Model for the AGV Case Study

8.2.2 Case Study Definition

The testing process considered in this case study is composed of three steps: build the model, define test purposes, and generate test cases. The first step is related to the definition of a TIOSTS representing the specification. Next, at the second step, test purposes are defined also using the TIOSTS formalism. Once the specification and test purposes have been defined, the test case generation and selection are automatically performed by the developed prototype tool.

The case study was conducted by only one tester with large experience in TIOSTS models. The case study aimed at testing two scenarios: an execution with no interruptions and another with one interruption. The first scenario was chosen for demonstrating that the strategy can be used for generating test cases with no interruptions. The latter is related to the most important scenario: an interruption occurs because an obstacle has been identified.

All information used as input for the execution of this case study was a high level description of the SUT described in two pages and the fault model defined in Subsection 8.1.2.

8.2.3 Case Study Results

This subsection presents and discusses the obtained results. Considering the first activity of the process described in Subsection 8.2.2, a TIOSTS representing the specification was defined (see Figure 8.8). At the beginning, the AGV system expects as input (represented by the action between locations $S1$ and $S2$ in Figure 8.8) the initial reference and path of the plan to be followed. After the input action, the AGV system emits several output actions: a message indicating that the file was successfully read (action between locations $S2$ and $S3$), another indicating that the references were successfully decoded (action between locations $S3$ and $S4$), and another message indicating that the path was successfully decoded (action between locations $S4$ and $S5$). After reading all needed information, the AGV system starts moving (action between locations $S5$ and $S6$). A periodic task is executed every 2000 milliseconds when the AGV system is moving. For controlling this task the *periodicClock* clock is used. Moreover, two interruptions can occur when the AGV system is moving: one related to the self diagnosis (Locations $I2.1$ and $I2.2$) and another related to the detection of an obstacle (Locations from $I1.1$ to $I1.4$). The *interruptionClock* clock is used to indicate that the latter interruption must be treated within at most 500 milliseconds. Finally, the AGV system emits an output message to indicate that the plan was successfully executed.

The test purpose depicted in Figure 8.9, named TP1, was defined for testing the scenario where no interruptions occur and the other test purpose of Figure 8.10, named TP2, was defined for testing the scenario where an obstacle is identified and an interruption occurs. All

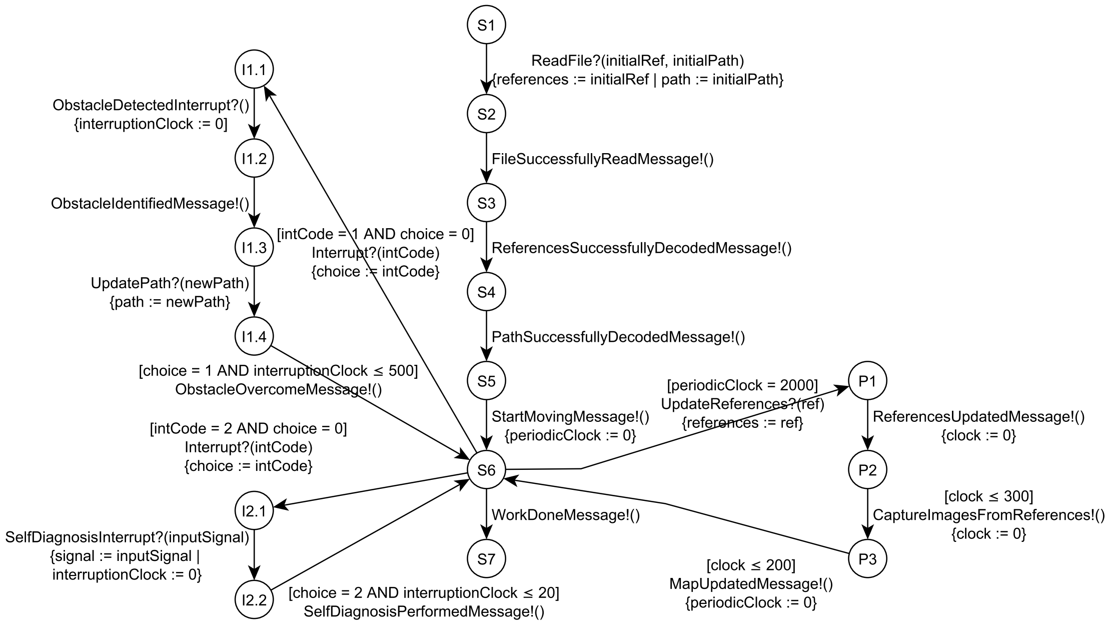

S1
ReadFile?(initialRef, initialPath)
{references := initialRef | path := initialPath}
S2
FileSuccessfullyReadMessage!()
S3
ReferencesSuccessfullyDecodedMessage!()
S4
PathSuccessfullyDecodedMessage!()
S5
StartMovingMessage!()
{periodicClock := 0}
S6
WorkDoneMessage!()
S7
I1.1
ObstacleDetectedInterrupt?()
{interruptionClock := 0]
I1.2
ObstacleIdentifiedMessage!()
I1.3
UpdatePath?(newPath)
{path := newPath}
I1.4
[intCode = 1 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
[choice = 1 AND interruptionClock ≤ 500]
ObstacleOvercomeMessage!()
[intCode = 2 AND choice = 0]
Interrupt?(intCode)
{choice := intCode}
I2.1
SelfDiagnosisInterrupt?(inputSignal)
{signal := inputSignal |
interruptionClock := 0}
I2.2
[choice = 2 AND interruptionClock ≤ 20]
SelfDiagnosisPerformedMessage!()
[periodicClock = 2000]
UpdateReferences?(ref)
{references := ref}
P1
ReferencesUpdatedMessage!()
{clock := 0}
P2
[clock ≤ 300]
CaptureImagesFromReferences!()
{clock := 0}
P3
[clock ≤ 200]
MapUpdatedMessage!()
{periodicClock := 0}

Figure 8.8: TIOSTS Specification for the AGV Case Study

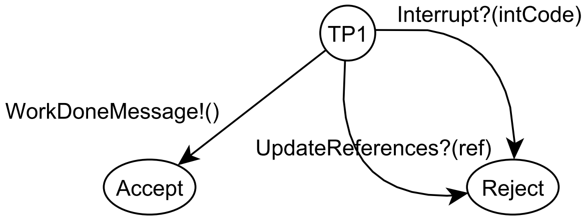


Figure 8.9: TIOSTS Test Purpose for the Scenario with no Interruptions (TP1)

TIOSTS models automatically generated by the prototype tool are presented in Appendix B.

Table 8.2 summarizes the metrics collected during the execution of all activities defined for this case study. Considering the first activity of the process, the tester spent 50 minutes to manually implement the TIOSTS representing the specification (Table 8.2, line 1).

Considering the scenario with no interruptions, the test purpose was defined in 4 minutes (Table 8.2, line 2). Once the specification and test purpose is implemented, the test case generation and selection was automatically performed in 3 seconds (Table 8.2, line 3) and only one test case was generated (Table 8.2, line 4).

The test purpose for the scenario with one interruption was defined in 3 minutes (Table 8.2, line 5). For this case, the test case generation and selection was performed in 3 seconds (Table 8.2, line 6) and only one test case was generated (Table 8.2, line 7).

Since all needed information was collected, the questions of the defined measurement

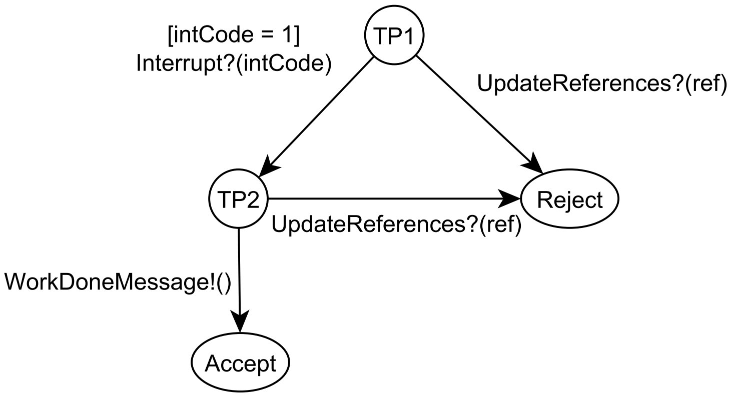


Figure 8.10: TIOSTS Test Purpose for the Scenario with One Interruption (TP2)

Table 8.2: Metrics of the AGV Case Study

	Metrics	**Time**
1	Definition and implementation of the TIOSTS specification	50 *min*
2	Definition and implementation of the TIOSTS test purpose TP1	4 *min*
3	Test case generation time considering TP1	3 *sec*
4	Number of Test Cases considering TP1	1
5	Definition and implementation of the TIOSTS test purpose TP2	3 *min*
6	Test case generation time considering TP2	3 *sec*
7	Number of Test Cases considering TP2	1
8	Execution time of all test process for all generated test cases	57.1 *min*
9	Fault model coverage	100%

model can be answered. For the first question, "what is the effort required to use this approach?", we have the following effort required to generate test cases for the two defined scenarios:

$$\begin{aligned} E &= E_1 + E_2 + E_3 \\ &= 50\,\mathit{min} + 4\,\mathit{min} + 3\,\mathit{min} + 3\,\mathit{sec} + 3\,\mathit{sec} \\ &= 57.1\,\mathit{min} \end{aligned}$$

The fault model profile defined in Subsection 8.1.2 was instantiated, based on the scenarios described in Subsection 8.2.2, in order to evaluate the effectiveness of the proposed approach.

- After the obstacle detection interruption, the AGV system does not maintain the data of the path to be followed;
- After the obstacle detection interruption, the AGV system does not finish its mission;
- After receiving initial references and the initial path, the SUT performs an unspecified output;
- An obstacle is overcome in more than 500 ms.

As the specification does not specify lower bounds as time requirements, it not possible to instantiate faults related to clock guard restrictions. As shown in Table 8.2, line 9, 100% of the defined fault model instance is covered by the defined test cases.

8.3 Concluding Remarks

This chapter presented two case studies performed to evaluate the applicability of the proposed approach. The first case study allowed to evaluate the test case generation and execution once an implementation is available. It was not possible to execute test cases for the second case study because there is no implementation available. Even thus, these case studies allowed to realize that the effort spent to generate test cases for checking specific scenarios is minimal when compared to the time spent to perform the entire process, even when interruptions are taken into account. Another strength of the work is the automation of some parts of the test process such as test case generation and evaluation of the results given that the logs have been generated. However, some points of improvements were identified such as the development of algorithms to translate the TIOSTS test case into C code in order to reduce the time spent to execute test cases.

Chapter 9

Concluding Remarks

This chapter summarizes the main results of this work and presents some suggestions for future work.

9.1 Conclusions

The main objective of this thesis was to provide an approach to conformance testing of real-time systems based on the use of a symbolic model that abstracts both time and data in order to broadening the application of conformance testing in this field. This thesis also presented a conformance testing theory to deal with the model proposed and described how test cases can be generated. Moreover, interruption testing of real-time systems was taken into account. For this, as a result of an initial investigation, an approach to conformance testing of non-real-time reactive systems with interruptions was proposed.

Considering the research questions defined in Chapter 1, the following results were achieved:

Research Question 1 *In which ways can we extend the symbolic model-based testing theory to be able to test real-time systems in an accurate manner?*

In order to answer this first research question a new symbolic model-based testing approach was proposed by combining symbolic transition systems [108] with timed automata [5]. Thus, the proposed model can handle both data and time requirements symbolically (see

Chapter 5). Furthermore, a conformance testing framework is proposed in Chapter 6 along with algorithms for test case generation.

Research Question 2 *In a real-time symbolic model-based testing context, how can we provide models to be able to specify and test asynchronous events such as interruptions in an accurate manner?*

In order to answer this second research question an initial investigation was performed in the context of non-real-time reactive systems. As a result, Chapter 3 presented a complete conformance testing approach for reactive systems. This work intended to investigate interruptions in a simple context. Considering the interruption testing of real-time systems, Chapter 7 presented a strategy based on the proposed symbolic model-based conformance testing approach to specify and test interruptions.

Research Question 3 *In a real-time symbolic model-based testing context, is it possible to provide an automated oracle?*

As discussed in Chapter 2, an oracle is a mechanism composed of a result generator and a comparator. In an automated oracle these two activities are fully automated. As the test cases generated by our approach describes all outputs allowed by specifications, the first activity of an oracle is automatically performed. According to what was discussed in Section 6.3, each execution of an implementation produces a log that describes the executed scenario. Next, this log is compared to the test case in order to emit a verdict. As this comparison is automatically performed (see Chapter 8), we can state that an automated oracle was provided by our approach. But it is important to remark that the test process is not completely automated, since TIOSTS test cases are manually translated into C code.

In summary, this thesis provides the following contributions:

1. An approach to conformance testing of reactive systems with interruptions and a tool to facilitate the practical application of the proposal;

2. A complete review of relevant work on conformance testing of real-time systems, described in Chapter 4, that resulted in the identification of some open problems.

3. A new conformance testing approach to real-time systems, where the SUT is modelled using a symbolic model that abstract both time and data;
4. A test case generation process based on symbolic execution and constraint solving for the data aspects combined with symbolic analysis of timed aspects.
5. A prototype tool implementing all algorithms of the test case generation process, which is essential in the generation of test cases from symbolic models.
6. A strategy to interruption testing of real-time systems along with a way of defining test purposes in order to check specific interruptions;
7. An initial test architecture including automatic ways of test execution and reliable verdicts achievements.
8. Results of case studies involving the use of the proposed work that show the feasibility of the practical application of the proposal.

Considering the related work presented in Chapter 4, all tables are presented here again in order to compare the related work with our approach. Table 9.1 shows that our approach generates test cases in an offline way using test purposes as test case selection strategy, there is tool support based on TIOSTS specification language, and quiescence is not taken into account. Table 9.2 shows that the tioco conformance relation is adopted and only deterministic specifications are taken into account along with input-complete implementations. Table 9.3 presents that our approach deals with analogue-time models and it is able to generate both instantiated and abstract test cases, considering that the CVC3 SMT Solver can be used to instantiate abstract test cases. Furthermore, our approach allows the specification of synchronous and asynchronous events and provides an automated oracle.

9.2 Future Work

With the completion of this work, there are several opportunities for future work. Next, some ideas are described:

Work	Test Case Generation	TP	Tool	Spec. Language	Quiesc.
Cardell-Oliver	offline	yes*	Essex*	TIOLTS	no
En-Nouaary et al.	offline	yes	no	deterministic and output urgent TAIO	no
Li et al.	offline	yes	no	RT Statecharts	no
Khoumsi	offline	yes	no	non-deterministic TIOSA	no
Briones and Brinksma	offline	no	no	TIOLTS	yes
Bohnenkamp and Belinfante	online	yes	yes*	non-deterministic safety TAIO	yes
Bodeveix et al.	offline	yes	no	a kind of TAIO	no
Larsen et al.	online	yes	TRON	TAIO (with guards on locations and transitions)	no
Hessel et al.	offline	yes	CoVer	deterministic and output urgent TAIO	no
Merayo et al.	offline	no	no	non-deterministic TEFSM	no
Krichen and Tripakis	offline and online	yes	TTG*	partially-observable and non-deterministic TAIO	no
Zheng et al.	offline	yes*	TROMLAB*	TEFSM	no
David et al.	offline	yes	TIGA	TIOGA	no
Adjir et al.	offline	yes	TINA	Prioritized Time Petri Nets	no
Styp et al.	no	no	no	STA	no
Timo et al.	offline	yes	no	VDTA	no
Andrade	offline	yes	yes	TIOSTS	no

Table 9.1: Comparison with Related Work

Take quiescence into account: in practice, tests observe the behaviour of the system and the absence of outputs. Then, an important future work is to extend the proposed conformance testing approach to deal with quiescence;

Take internal actions into account: although the TIOSTS definition allows the specification of internal actions, the developed algorithms do not treat internal actions;

Deal with non-input-complete implementations: practically all reviewed approaches assume the input-completeness of the implementation. the intention of this future work is to investigate how this assumption could be discarded;

Deal with non-deterministic models: the intention here is to extend the approach proposed in this thesis to deal with non-deterministic models;

Propose generic fault models: fault models based on approaches that use region clocks such as the work presented in [48; 1] cannot be completely reused in a context where zones are used because of the higher abstraction level. Then, this issue must be better investigated;

Generate test cases based on coverage criteria: this thesis proposes a test selection approach based on test purposes, but there are other approaches to test case selection such as selection based on coverage criteria;

Deal with more data types: investigate techniques such as abstract interpretation in order to deal with more data types;

Improve the test execution activity: the test case execution can be improved by automating the translation of TIOSTS test cases into C code. In this case, the time spent in the test case execution activity can be decreased;

Execution of test cases in other platforms: the intention is to extend the work proposed in this thesis to allow the execution of test cases in other environments besides FreeRTOS;

Develop a fully integrated environment for testing real-time systems: it is important to integrate the test case generation tool with the test execution tools in order to provide a complete environment to generate and execute test cases;

Perform new case studies: since a complete environment for test case generation and execution is proposed as a future work, it is essential to perform new case studies in order to evaluate the applicability of the work.

Work	Conf. Relation	Specification	Implementation
Cardell-Oliver	trace equivalence	input-complete and must have more states than the implementation.	input-complete
En-Nouaary et al.	trace equivalence	input-complete and must have the same number of locations as the implementation.	input-complete
Li et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Khoumsi	timed trace inclusion	input-complete	input-complete
Briones and Brinksma	ioco with quiescence	input-complete	input-complete*
Bohnenkamp and Belinfante	ioco with quiescence	input-complete	input-complete
Bodeveix et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
Larsen et al.	timed trace inclusion	deterministic and input-complete	input-complete
Hessel et al.	timed trace inclusion	deterministic, input-complete, and output urgent	input-complete
Merayo et al.	there are several conformance relations	input-complete	input-complete
Krichen and Tripakis	tioco	no restriction on input-completeness	input-complete
Zheng et al.	no conformance relation is defined	assumptions are not discussed	assumptions are not discussed
David et al.	tioco	input-complete	input-complete
Adjir et al.	timed trace inclusion	deterministic, input-complete, and output urgent	deterministic, input-complete, and output urgent
Styp et al.	stioco	non-deterministic	input-complete
Timo et al.	tvco	assumptions are not discussed	assumptions are not discussed
Andrade	tioco	deterministic and no restriction on input-completeness	input-complete

Table 9.2: Comparison with Related Work

Work	**Time**	**Test Cases**	**Communication**	**Oracle**
Cardell-Oliver	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
En-Nouaary et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Li et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Khoumsi	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Briones and Brinksma	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Bohnenkamp and Belinfante	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Bodeveix et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Larsen et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	automated
Hessel et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Merayo et al.	digital-time model	instantiated	synchronous	partial
Krichen and Tripakis	digital and analogue-time models*	instantiated	synchronous	automated
Zheng et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
David et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Adjir et al.	analogue-time model (internally the model is digitised)	instantiated	synchronous	partial
Styp et al.	analogue-time model	undefined	synchronous	undefined
Timo et al.	analogue-time model	undefined	synchronous	undefined
Andrade	analogue-time model	instantiated and abstract	synchronous and asynchronous	automated

Table 9.3: Comparison with Related Work

Bibliography

[1] M. S. AbouTrab and S. Counsell. Fault coverage measurement of a timed test case generation approach. In *ECBS '10: Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 141–149, Washington, DC, USA, 2010. IEEE Computer Society.

[2] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[3] Noureddine Adjir, Pierre Saqui-Sannes, and Kamel Mustapha Rahmouni. Testing real-time systems using tina. In *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, TESTCOM '09/FATES '09, pages 1–15, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Noureddine Adjir, Pierre Saqui-Sannes, and Kamel Mustapha Rahmouni. Time-optimal real-time test case generation using prioritized time petri nets. In *VALID '09: Proceedings of the First International Conference on Advances in System Testing and Validation Lifecycle*, pages 110–116, 2009.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[6] Wilkerson L. Andrade. Interaction test case generation for mobile phone applications. Master's thesis, Federal University of Campina Grande, Mar 2007.

[7] Wilkerson L. Andrade and Patrícia D. L. Machado. Modeling and testing interruptions in reactive systems using symbolic models. In *SAST'08: Proc. of the 2nd Brazilian*

Work. on Systematic and Automated Software Testing, pages 34–43, Porto Alegre, 2008. SBC.

[8] Wilkerson L. Andrade and Patrícia D. L. Machado. Interruption testing of reactive systems. In *Formal Methods: Foundations and Applications*, volume 5902 of *LNCS*, pages 37–53. Springer, 2009.

[9] Wilkerson L. Andrade and Patrícia D. L. Machado. Interruption testing of reactive systems. *Formal Aspects of Computing*, pages 1–23, 2011. To appear.

[10] Wilkerson L. Andrade, Patrícia D. L. Machado, Everton L. G. Alves, and Diego R. Almeida. Test case generation of embedded real-time systems with interruptions for FreeRTOS. In *Formal Methods: Foundations and Applications*, volume 5902 of *LNCS*, pages 54–69. Springer, 2009.

[11] Wilkerson L. Andrade, Patrícia D. L. Machado, Thierry Jéron, and Hervé Marchand. Abstracting time and data for conformance testing of real-time systems. In *A-MOST '11: Proceedings of the 7th Workshop on Advances in Model Based Testing*, March 2011. To appear.

[12] Wilkerson L. Andrade, Francisco G. O. Neto, and Patrícia D. L. Machado. Geração de casos de teste de interrupção para aplicações de celulares. In *WTF '07: Proc. of the VIII Test and Fault Tolerance Workshop*, pages 129–142, Porto Alegre, RS, Brazil, 2007. Brazilian Computer Society.

[13] George S. Avrunin, James C. Corbett, and Laura K. Dillon. Analyzing partially-implemented real-time systems. *IEEE Trans. Softw. Eng.*, 24(8):602–614, 1998.

[14] Victor R. Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, University of Maryland at College Park, College Park, MD, USA, 1992.

[15] Gilles Bernot. Testing against formal specifications: a theoretical view. In *TAPSOFT '91: Vol. 2*, pages 99–119, New York, NY, USA, 1991. Springer-Verlag.

[16] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, Mar 1991.

[17] N. Bertrand, T. Jéron, A. Stainer, and M. Krichen. Off-line test selection with test purposes for non-deterministic timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, march 2011. To appear.

[18] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, Boston, MA, USA, 1999.

[19] Jean-Paul Bodeveix, Rachid Bouaziz, and Ousmane Koné. Test method for embedded real-time systems. In *ERCIM European Workshop on Dependable Software Intensive Embedded Systems*, pages 1–10, Porto, Portugal, 2005. ERCIM.

[20] Henrik Bohnenkamp and Axel Belinfante. Timed testing with TorX. In *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188, Newcastle, UK, 2005. Springer-Verlag.

[21] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 264–279. Springer, 1998.

[22] Ahmed Bouajjani, Yassine Lakhnech, and Sergio Yovine. Model-checking for extended timed temporal logics. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 306–326, London, UK, 1996. Springer-Verlag.

[23] Laura Brandán Briones and Ed Brinksma. A test generation framework for *quiescent* real-time systems. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 64–78. Springer, 2005.

[24] Laura Brandán Briones and Ed Brinksma. Testing real-time multi input-output systems. In *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 264–279. Springer-Verlag, 2005.

[25] Renée C. Bryce and Charles J. Colbourn. Test prioritization for pairwise interaction coverage. In *A-MOST '05: Proceedings of the first international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[26] Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, 2008.

[27] Jens R. Calamé, Natalia Ioustinova, and Jaco van de Pol. Automatic model-based generation of parameterized test cases using data abstraction. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proc. of the Doctoral Symposium affiliated with the Fifth Integrated Formal Methods Conference (IFM 2005)*, volume 191 of *Electronic Notes in Computer Science*, pages 25–48. Elsevier, October 2007.

[28] Rachel Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, Dec. 2000.

[29] Emanuela G. Cartaxo, Wilkerson L. Andrade, Francisco G. O. Neto, and Patrícia D. L. Machado. LTSBT: A tool to generate and select functional test cases for embedded systems. In *SAC'08: Proc. of the 2008 ACM symposium on Applied computing*, volume 2, pages 1540–1544, New York, NY, USA, 2008. ACM Press.

[30] Emanuela G. Cartaxo, Patrícia D. L. Machado, and Francisco G. Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 2009. Early Online View: http://dx.doi.org/10.1002/stvr.413.

[31] Albert M. K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[32] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In *TACAS'02: Proc. of the Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 151–173. Springer, 2002.

[33] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[34] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Camille Constant, Thierry Jéron, Hervé Marchand, and Vlad Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Trans. Software Eng.*, 33(8):558–574, 2007.

[36] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[37] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. A game-theoretic approach to real-time system testing. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 486–491, New York, NY, USA, 2008. ACM.

[38] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Timed testing under partial observability. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 61–70, Washington, DC, USA, 2009. IEEE Computer Society.

[39] André L. L. de Figueiredo, Wilkerson L. Andrade, and Patrícia D. L. Machado. Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006. Proceedings of the AMOST'2006.

[40] R. G. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.

[41] R. G. de Vries and J. Tretmans. Towards formal test purposes. In *Proceedings of 1st International Workshop on Formal Approaches to Testing of Software 2001 (FATES'01)*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denmark, August 2001.

[42] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1990.

[43] I. K. El-Far and J. A. Whittaker. Model-based software testing. *Encyclopedia on Software Engineering*, 2001.

[44] Abdeslam En-Nouaary. A scalable method for testing real-time systems. *Software Quality Control*, 16:3–22, March 2008.

[45] Abdeslam En-Nouaary and Rachida Dssouli. A guided method for testing timed input output automata. In Dieter Hogrefe and Anthony Wiles, editors, *Testing of Communicating Systems*, volume 2644 of *Lecture Notes in Computer Science*, pages 211–225. Springer Berlin / Heidelberg, 2003. Proceedings of the 15th IFIP international conference on Testing of communicating systems (TestCom'03).

[46] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 28(11):1023–1038, 2002.

[47] Abdeslam En-Nouaary, Rachida Dssouli, Ferhat Khendek, and Abdelkader Elqortobi. Timed test cases generation based on state characterization technique. In *RTSS '98: Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 220–230, Washington, DC, USA, 1998. IEEE Computer Society.

[48] Abdeslam En-Nouaary, Ferhat Khendek, and Rachida Dssouli. Fault coverage in testing real-time systems. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 150, Washington, DC, USA, 1999. IEEE Computer Society.

[49] ETSI. *European Standard (ES) 201 873 - The Testing and Test Control Notation Version 3 (TTCN-3), Part 1: TTCN-3 Core Language, Part 2: Tabular Presentation Format for TTCN-3 (TFT), Part 3: Graphical Presentation Format for TTCN-3 (GFT), Part 4: Operational Semantics, Part 5: The TTCN-3 Runtime Interface (TRI), Part*

6: The TTCN-3 Control Interfaces (TCI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 2005.

[50] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property oriented test case generation. In *Formal Approaches to Software Testing, Proceedings of FATES 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 147–163, Montreal, Canada, 2004. Springer.

[51] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, 1997.

[52] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer, 2006.

[53] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *Testing of Communicating Systems*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

[54] L.D. Gowen. Specifying and verifying safety-critical software systems. *Proceedings of the IEEE Seventh Symposium on Computer-Based Medical Systems*, pages 235–240, Jun 1994.

[55] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer, 1993.

[56] Dick Hamlet. Software quality, software process, and software testing. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 41, pages 191–229. Academic Press, 1995.

[57] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, 2004.

[58] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111:193–244, June 1994.

[59] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M.

Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 77–117. Springer, 2008.

[60] Anders Hessel, Kim Guldstrand Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using UPPAAL. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *LNCS*, pages 114–130. Springer, 2004.

[61] Anders Hessel and Paul Pettersson. A test case generation algorithm for real-time systems. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 268–273, Washington, DC, USA, 2004. IEEE Computer Society.

[62] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[63] Bertrand Jeannet, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Symbolic test selection based on approximate analysis. In *TACAS'05: Proc. of Int. Conf. on Tools and Alg. for Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 349–364, 2005.

[64] Thierry Jéron. Symbolic model-based test selection. *Electron. Notes Theor. Comput. Sci.*, 240:167–184, July 2009. Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008).

[65] Thierry Jéron, Hervé Marchand, and Vlad Rusu. Symbolic determinisation of extended automata. In *Proceedings of the 4th IFIP International Conference on Theoretical Computer Science*, volume 209 of *IFIP book series*, pages 197–212. Springer-Verlag, 2006.

[66] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving. In *ICST '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, pages 479–488, Washington, DC, USA, 2010. IEEE Computer Society.

[67] Paul Jorgensen. *Software Testing: A Craftman's Approach*. CRC Press, Inc., 3rd edition, 2008.

[68] D. S. Jovanovic, B. Orlic, and J. F. Broenink. On issues of constructing an exception handling mechanism for CSP-based process-oriented concurrent software. In *Proceedings of Communicating Process Architectures CPA 2005*, pages 18–21, Eindhoven, NL, 2005. IOS Press.

[69] Ahmed Khoumsi. Complete test graph synthesis for symbolic real-time systems. *ENTCS*, 130:79–100, 2005.

[70] Ahmed Khoumsi. On synthesizing test cases in symbolic real-time testing. *Journal of the Brazilian Computer Society*, 12:31–48, 2007.

[71] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[72] Pieter Koopman and Rinus Plasmeijer. Testing reactive systems with GAST. In Stephen Gilmore, editor, *Trends in Functional Programming*, volume 4 of *Trends in Functional Programming*, pages 111–129. Intellect, 2003.

[73] Moez Krichen. *Model-Based Testing for Real-Time Systems*. PhD thesis, Université Joseph Fourier, Dec 2007.

[74] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *SPIN'04: Proc. of the 11th Int. SPIN Workshop on Model Checking of Software*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004.

[75] Moez Krichen and Stavros Tripakis. Real-time testing with timed automata testers and coverage criteria. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 134–151. Springer-Verlag, 2004.

[76] Moez Krichen and Stavros Tripakis. An expressive and implementable formal framework for testing real-time systems. In *TestCom'05: Proc. of the 17th IFIP Int. Conf. on Testing of Communicating Systems*, volume 3502 of *LNCS*, pages 209–225. Springer, 2005.

[77] Moez Krichen and Stavros Tripakis. Interesting properties of the real-time conformance relation tioco. In *ICTAC'06: Proc. of the 3rd Int. Colloquium on Theoretical Aspects of Computing*, volume 4281 of *LNCS*, pages 317–331. Springer, 2006.

[78] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[79] Phillip A. Laplante. *Real-Time System Design and Analysis.* John Wiley & Sons, 2004.

[80] Kim Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 79–94. Springer, 2005.

[81] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306, New York, NY, USA, 2005. ACM.

[82] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[83] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes: Adapting the notion of specification to testing. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, pages 127–134, Washington, DC, USA, 2001. IEEE Computer Society.

[84] Daniel Leitao, Dante Torres, and Flávia Barros. NLForSpec: Translating natural language descriptions into formal test case specifications. In *Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2007)*, pages 129–134, Boston, Massachusetts, USA, 2007. Knowledge Systems Institute Graduate School.

[85] Grégory Lestiennes and Marie-Claude Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *ISSRE'02: Proc. of the 13th Int. Symp. on Software Reliability Engineering*, page 3. IEEE Computer Society, 2002.

[86] Qing Li and Carolyn Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

[87] Shuhao Li, Ji Wang, Wei Dong, and Zhi-Chang Qi. Property-oriented testing of real-time systems. In *APSEC'04: Proc. of the 11th Asia-Pacific Software Engineering Conference*, pages 358–365. IEEE Computer Society, 2004.

[88] L. Lorentsen, A.-P. Tuovinen, and J. Xu. Modelling feature interactions in mobile phones. In *Feature Interaction in Composed Systems (ECOOP 2001)*, pages 7–13, Budapest, Hungary, 2001.

[89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, sep 1989.

[90] Augusto Q. Macedo, Wilkerson L. Andrade, Diego R. Almeida, and Patrícia D. L. Machado. Automating test case execution for real-time embedded systems. In *ICTSS'10: Proceedings of the 22nd IFIP International Conference on Testing Software and Systems*, pages 37–42, 2010. Short Paper.

[91] Patrícia D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, LFCS, University of Edinburgh, UK, 2000.

[92] Patrícia D. L. Machado and Wilkerson L. Andrade. The oracle problem for testing against quantified properties. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pages 415–418, Washington, DC, USA, 2007. IEEE Computer Society.

[93] Patrícia D. L. Machado and Augusto C. A. Sampaio. Automatic test-case generation. In Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *Testing Techniques in Software Engineering*, volume 6153 of *Lecture Notes in Computer Science*, pages 59–103. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

[94] Patrícia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards property oriented testing. *Electronic Notes in Theoretical Computer Science*, 184:3–19, 2007.

[95] John D. McGregor and David A. Sykes. *A practical guide to testing object-oriented software*. Addison-Wesley, Boston, MA, USA, 2001.

[96] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. Formal testing of systems presenting soft and hard deadlines. In *FSEN'07: Proc. of the Int. Symp. on Fundamentals of Software Engineering*, volume 4767 of *LNCS*, pages 160–174. Springer, 2007.

[97] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. Extending EFSMs to specify and test timed systems with action durations and time-outs. *IEEE Trans. Comput.*, 57(6):835–844, 2008.

[98] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. Formal testing from timed finite state machines. *Comput. Netw.*, 52(2):432–460, 2008.

[99] Mercedes G. Merayo, Manuel Núñez, and Ismael Rodríguez. A formal framework to test soft and hard deadlines in timed systems. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2011.

[100] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 396–397, Washington, DC, USA, 2004. IEEE Computer Society.

[101] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2nd edition, 2004.

[102] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from CSP models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 258–273, Berlin, Heidelberg, 2008. Springer-Verlag.

[103] Manuel Núñez and Ismael Rodríguez. Conformance testing relations for timed systems. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Software Testing*, volume 3997 of *Lecture Notes in Computer Science*, pages 103–117. Springer Berlin / Heidelberg, 2006.

[104] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11:339–353, October 2009.

[105] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2007.

[106] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., 7th edition, 2009.

[107] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988.

[108] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In *IFM'00: Proc. of the Second Int. Conf. on Integrated Formal Methods*, pages 338–357. Springer, 2000.

[109] Steve Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[110] Ian Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, Boston, MA, USA, 9th edition, 2010.

[111] Mitsuo Takaki, Diego Cavalcanti, Rohit Gheyi, Juliano Iyoda, Marcelo D'Amorim, and Ricardo B. Prudêncio. Randomized constraint solvers: a comparative study. *Innov. Syst. Softw. Eng.*, 6:243–253, September 2010.

[112] The FreeRTOS.org Project. FreeRTOS. http://www.freertos.org.

[113] Omer Nguena Timo, Hervé Marchand, and Antoine Rollet. Automatic test generation for data-flow reactive systems with time constraints. In *ICTSS'10: Proceedings of the 22nd IFIP International Conference on Testing Software and Systems*, pages 25–30, 2010. Short Paper.

[114] Omer Nguena Timo and Antoine Rollet. Conformance testing of variable driven automata. In *WFCS'10: Proceedings of the 8th IEEE International Workshop on Factory Communication Systems*, pages 241–248. IEEE Computer Society, 2010.

[115] Dante Torres, Daniel Leitao, and Flávia Barros. Motorola SpecNL: A hybrid system to generate nl descriptions from test case specifications. In *HIS '06: Proceedings of the*

Sixth International Conference on Hybrid Intelligent Systems, page 45, Washington, DC, USA, 2006. IEEE Computer Society.

[116] Jan Tretmans. Conformance testing with labelled transition systems: implementation relations and test generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, 1996.

[117] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In *TACAS'96: Proc. of the Second Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 127–146. Springer, 1996.

[118] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99: Proc. of the 10th Int. Conf. on Concurrency Theory*, pages 46–65. Springer, 1999.

[119] Jan Tretmans and Ed Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proceedings of the First European Conference on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, 2003.

[120] Sabrina von Styp, Henrik Bohnenkamp, and Julien Schmaltz. A conformance testing relation for symbolic timed automata. In Krishnendu Chatterjee and Thomas Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *Lecture Notes in Computer Science*, pages 243–255. Springer Berlin / Heidelberg, 2010.

[121] Rob Williams. *Real-Time Systems Development*. Butterworth-Heinemann, Oxford, UK, 2006.

[122] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. *Form. Methods Syst. Des.*, 11:113–136, August 1997.

[123] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 243–258, London, UK, UK, 1995. Chapman & Hall, Ltd.

[124] Mao Zheng, Vasu Alagar, and Olga Ormandjieva. Automated generation of test suites from formal specifications of real-time reactive systems. *J. Syst. Softw.*, 81(2):286–304, 2008.

Appendix A

Proofs

Proof of Theorem 3.1.

Proof of soundness:

According to Definition 3.7 a test suite is sound if all of its test cases are sound. Furthermore, a test case TC is sound for S and **conf** if $\forall$SUT,

$$\text{SUT } \textbf{conf } S \Rightarrow \neg(TC \text{ may reject SUT}).$$

Using the contraposition principle, we need to prove that if a test case TC may reject a SUT (implementing the specification S), then $\neg$(SUT **conf** S). Thus, we need to prove that $\forall$SUT,

$$TC \text{ may reject SUT} \Rightarrow \neg(\text{SUT } \textbf{conf } S).$$

By Definition 3.6 we need to prove that $\forall$SUT,

$$\exists \sigma \in \text{Traces}(TC \mid\mid \text{SUT}) : \text{verdict}(\sigma) = \textit{Fail} \Rightarrow \neg(\text{SUT } \textbf{conf } S).$$

Let SUT be an arbitrary implementation such that $\exists \sigma \in \text{Traces}(TC \mid\mid \text{SUT}) : \text{verdict}(\sigma) = \textit{Fail}$. Then, let $\sigma = [a_1]\omega_1\ [a_2]\omega_2\ \ldots\ [a_n]\omega_n \in ([A]L)^*$ be the trace corresponding to the interaction between TC and SUT until the verdict *Fail* is emitted. Also, let $\sigma_{n-1} = [a_1]\omega_1\ [a_2]\omega_2\ \ldots\ [a_{n-1}]\omega_{n-1}$ be a trace excluding $[a_n]\omega_n$. According to the verdicts definition (page 16), if a *Fail* is emitted then $[a_n]\omega_n$ is an output action. Thus, $Out(\text{SUT } \textit{after } \sigma_{n-1}) \neq \emptyset$ because $[a_n]\omega_n \in Out(\text{SUT } \textit{after } \sigma_{n-1})$.

Since *Fail* is obtained, $[a_n]\omega_n \notin Out(S \textit{ after } \sigma_{n-1})$. Hence, $Out(\text{SUT } \textit{after } \sigma_{n-1}) \nsubseteq Out(S \textit{ after } \sigma_{n-1})$ and consequently $\neg$(SUT **conf** S).

Then, $\forall$SUT, TC may reject SUT $\Rightarrow \neg$(SUT **conf** S) and, consequently,

$$\forall TC\ \forall \text{SUT},\ TC \text{ may reject SUT} \Rightarrow \neg(\text{SUT } \textbf{conf}\ S).$$

Proof of exhaustiveness:

For proving that the test suites generated by LTS-BT are exhaustive, we need to prove that for every non-conforming SUT there is a test purpose TP and a way of generating a test case TC from S and TP, such that TC may reject SUT.

According to Definition 3.7 a test suite is exhaustive for S and **conf** if $\forall$SUT,

$$\neg(\text{SUT } \textbf{conf}\ S) \Rightarrow \exists TC : TC \text{ may reject SUT.}$$

By Definition 3.3, if $\neg$(SUT **conf** S) then there is a trace $\sigma = [a_1]\omega_1\ [a_2]\omega_2\ \ldots\ [a_{n-1}]\omega_{n-1}\ [a_n]\omega_n \in \mathit{Traces}(S)$ and an output action $[a_{n+1}]\omega_{n+1} \in [A \setminus \{steps, conditions, beginInterruption\_X\}]L_O$ such that

$$[a_{n+1}]\omega_{n+1} \in Out(\text{SUT } \mathit{after}\ \sigma) \text{ and } [a_{n+1}]\omega_{n+1} \notin Out(S\ \mathit{after}\ \sigma).$$

Let $[a'_{n+1}]\omega'_{n+1}$ be the correct output action such that $[a'_{n+1}]\omega'_{n+1} \in Out(S\ \mathit{after}\ \sigma)$. Thus, σ and $[a'_{n+1}]\omega'_{n+1}$ can be used to define the following TP:

$$\text{“}\omega_1\ \omega_2\ \ldots\ \omega_{n-1}\ \omega_n; \omega'_{n+1}; \text{Accept”}.$$

Finally, a test case TC is generated based on S and the defined TP. Thus, during the test case execution the SUT produces $[a_{n+1}]\omega_{n+1}$ instead of $[a'_{n+1}]\omega'_{n+1}$. In this case, a *Fail* verdict is emitted as expected. Hence, TC may reject SUT according to Definition 3.6. □

Proof of Theorem 6.1.

Proof of soundness:

Let $[\![TC]\!] = \langle S, S^0, Act, T\rangle$ be the TIOLTS semantics of the test case $TC = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$. According to Definition 6.9 a test suite is sound if all of its test cases are sound. Furthermore, a test case TC is sound for $\mathcal{S}$ and **tioco** if $\forall \mathcal{I}$,

$$\mathcal{I}\ \textbf{tioco}\ \mathcal{S} \Rightarrow \neg(TC\ \textit{may reject}\ \mathcal{I}).$$

Using the contraposition principle, we need to prove that if a test case TC may reject $\mathcal{I}$ (implementing the specification $\mathcal{S}$), then $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$. Thus, we need to prove that $\forall \mathcal{I}$,

$$TC\ \textit{may reject}\ \mathcal{I} \Rightarrow \neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}).$$

By Definition 6.8 we need to prove that $\forall \mathcal{I}$,

$$\exists \sigma \in \textit{Traces}([\![TC]\!] \mid\mid \textit{ObservableTraces}(\mathcal{I})) : \textit{verdict}(\sigma) = \textbf{Fail} \Rightarrow \neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}).$$

Let $\mathcal{I}$ be an arbitrary implementation such that $\exists \sigma \in \textit{Traces}([\![TC]\!] \mid\mid \textit{ObservableTraces}(\mathcal{I})) : \textit{verdict}(\sigma) = \textbf{Fail}$. Then, let $\sigma = a_1\ a_2\ \ldots\ a_n \in (Act \backslash \Lambda^\tau)^*$ be the trace corresponding to the interaction between $[\![TC]\!]$ and an observable behaviour of $\mathcal{I}$ until the verdict **Fail** is emitted. Also, let $\sigma_{n-1} = a_1\ a_2\ \ldots\ a_{n-1}$ be a trace excluding a_n. According to the verdicts definition (Section 6.3), if a **Fail** is emitted then a_n is either an output action or a time-elapsing action. Thus, $Out(\mathcal{I}\ \textit{after}\ \sigma_{n-1}) \neq \emptyset$ because $a_n \in Out(\mathcal{I}\ \textit{after}\ \sigma_{n-1})$.

Since **Fail** is obtained, $a_n \notin Out(\mathcal{S}\ \textit{after}\ \sigma_{n-1})$. Hence, $Out(\mathcal{I}\ \textit{after}\ \sigma_{n-1}) \nsubseteq Out(\mathcal{S}\ \textit{after}\ \sigma_{n-1})$ and consequently $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$.

Then, $\forall \mathcal{I}$, $TC\ \textit{may reject}\ \mathcal{I} \Rightarrow \neg(\mathcal{I}$ **tioco** $\mathcal{S})$ and, consequently,

$$\forall TC\ \forall \mathcal{I},\ TC\ \textit{may reject}\ \mathcal{I} \Rightarrow \neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}).$$

Proof of exhaustiveness:

For proving that the test suites generated by our approach are exhaustive, we need to prove that for every non-conforming $\mathcal{I}$ there is a test purpose TP and a way of generating a test case TC from $\mathcal{S}$ and TP, such that TC may reject $\mathcal{I}$.

According to Definition 6.9 a test suite is exhaustive for $\mathcal{S}$ and **tioco** if $\forall \mathcal{I}$,

$$\neg(\mathcal{I}\ \textbf{tioco}\ \mathcal{S}) \Rightarrow \exists TC : TC\ \textit{may reject}\ \mathcal{I}.$$

By Definition 6.2, if $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$ then there is a trace $\sigma = a_1\ a_2\ \ldots\ a_{n-1}\ a_n \in$ *ObservableTraces*$(\mathcal{S})$ and an output event a_{n+1} (i.e., an output action or time-elapsing action) such that

$$a_{n+1} \in Out(\mathcal{I}\ \textit{after}\ \sigma) \text{ and } a_{n+1} \notin Out(\mathcal{S}\ \textit{after}\ \sigma).$$

Let a'_{n+1} be the correct output event such that $a'_{n+1} \in Out(\mathcal{S}\ \textit{after}\ \sigma)$. Thus, σ and a'_{n+1} can be used to define a TP with the path "$a_1\ a_2\ \ldots\ a_{n-1}\ a_n\ a'_{n+1}$" leading to an *Accept* location.

Finally, a test case TC is generated based on $\mathcal{S}$ and the defined TP. Thus, during the test case execution $\mathcal{I}$ produces a_{n+1} instead of a'_{n+1}. In this case, a **Fail** verdict is emitted as expected. Hence, TC may reject $\mathcal{I}$ according to Definition 6.8. □

Appendix B

TIOSTS Models

This appendix presents all TIOSTS models automatically generated by the prototype tool during the execution of the case studies described in Chapter 8.

B.1 TIOSTS Models of the Burglar Alarm System Case Study

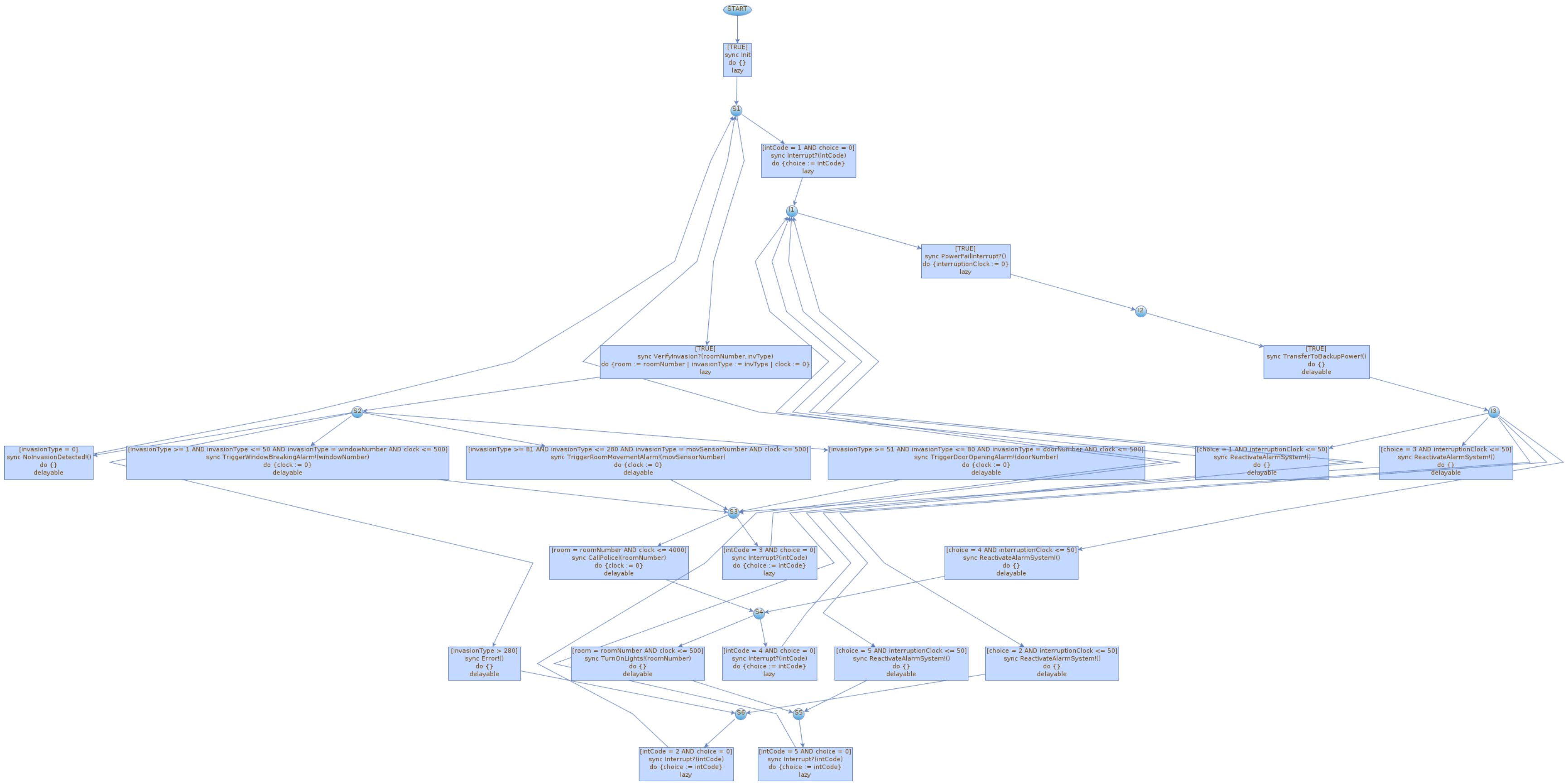

START
[TRUE]
sync Init
do {}
lazy
S1
[intCode = 1 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
I1
[TRUE]
sync PowerFailInterrupt?()
do {interruptionClock := 0}
lazy
I2
[TRUE]
sync TransferToBackupPower!()
do {}
delayable
I3
[TRUE]
sync VerifyInvasion?(roomNumber,invType)
do {room := roomNumber | invasionType := invType | clock := 0}
lazy
S2
[invasionType = 0]
sync NoInvasionDetected!()
do {}
delayable
[invasionType >= 1 AND invasionType <= 50 AND invasionType = windowNumber AND clock <= 500]
sync TriggerWindowBreakingAlarm!(windowNumber)
do {clock := 0}
delayable
[invasionType >= 81 AND invasionType <= 280 AND invasionType = movSensorNumber AND clock <= 500]
sync TriggerRoomMovementAlarm!(movSensorNumber)
do {clock := 0}
delayable
[invasionType >= 51 AND invasionType <= 80 AND invasionType = doorNumber AND clock <= 500]
sync TriggerDoorOpeningAlarm!(doorNumber)
do {clock := 0}
delayable
[choice = 1 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
[choice = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
S3
[room = roomNumber AND clock <= 4000]
sync CallPolice!(roomNumber)
do {clock := 0}
delayable
[intCode = 3 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
[choice = 4 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
S4
[invasionType > 280]
sync Error!()
do {}
delayable
[room = roomNumber AND clock <= 500]
sync TurnOnLights!(roomNumber)
do {}
delayable
[intCode = 4 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
[choice = 5 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
[choice = 2 AND interruptionClock <= 50]
sync ReactivateAlarmSystem!()
do {}
delayable
S6
S5
[intCode = 2 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
[intCode = 5 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy

Figure B.1: Burglar Alarm System Specification

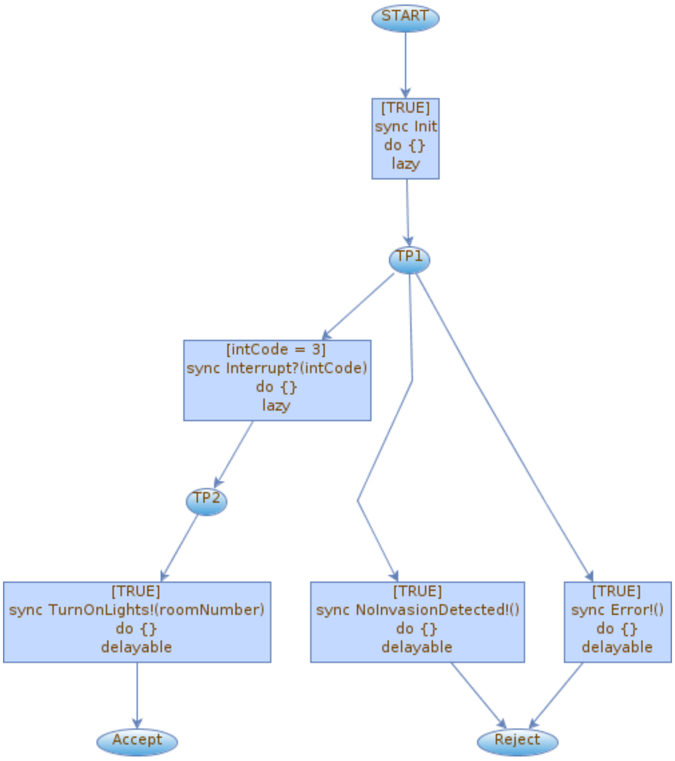


Figure B.2: Test Purpose

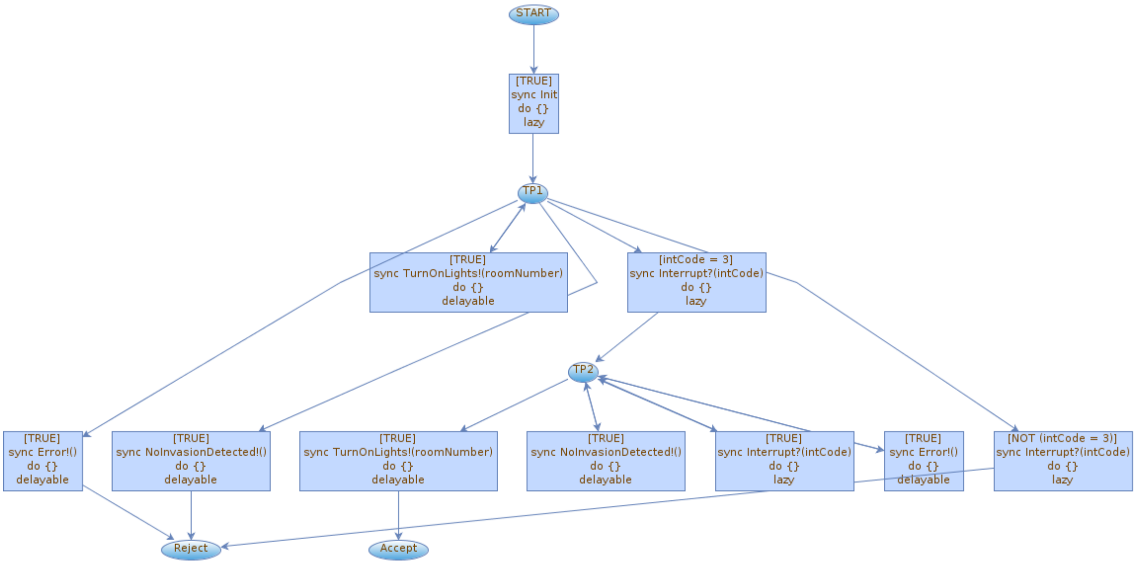


Figure B.3: Completed Test Purpose

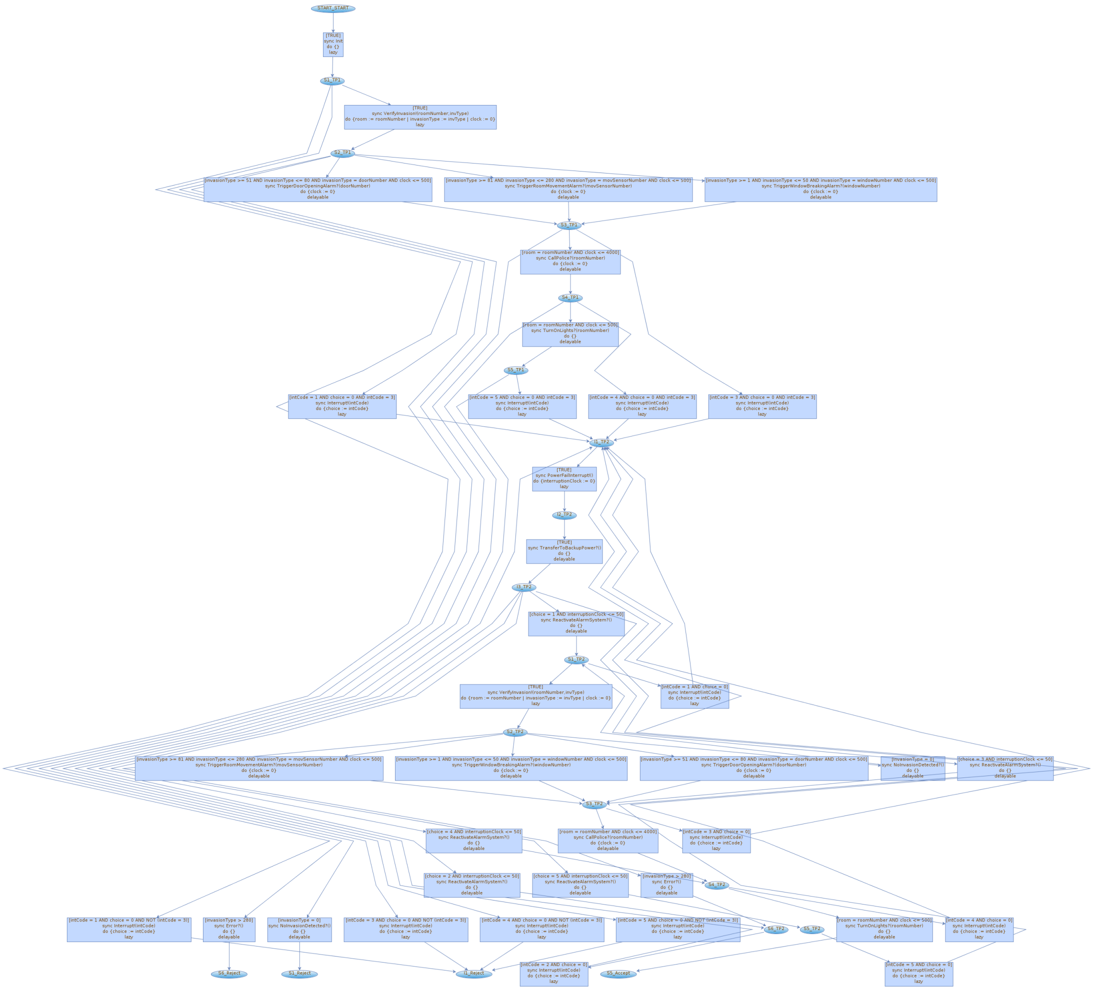

START_START
[TRUE] sync Init do {} lazy
S1_TP1
[TRUE] sync VerifyInvasion!(roomNumber,invType) do {room := roomNumber | invasionType := invType | clock := 0} lazy
S2_TP1
[invasionType >= 51 AND invasionType <= 80 AND invasionType = doorNumber AND clock <= 500] sync TriggerDoorOpeningAlarm?(doorNumber) do {clock := 0} delayable
[invasionType >= 81 AND invasionType <= 280 AND invasionType = movSensorNumber AND clock <= 500] sync TriggerRoomMovementAlarm?(movSensorNumber) do {clock := 0} delayable
[invasionType >= 1 AND invasionType <= 50 AND invasionType = windowNumber AND clock <= 500] sync TriggerWindowBreakingAlarm?(windowNumber) do {clock := 0} delayable
S3_TP1
[room = roomNumber AND clock <= 4000] sync CallPolice?(roomNumber) do {clock := 0} delayable
S4_TP1
[room = roomNumber AND clock <= 500] sync TurnOnLights?(roomNumber) do {} delayable
S5_TP1
[intCode = 1 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 5 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 4 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 3 AND choice = 0 AND intCode = 3] sync Interrupt!(intCode) do {choice := intCode} lazy
I1_TP2
[TRUE] sync PowerFailInterrupt!() do {interruptionClock := 0} lazy
I2_TP2
[TRUE] sync TransferToBackupPower?() do {} delayable
I3_TP2
[choice = 1 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
S1_TP2
[TRUE] sync VerifyInvasion!(roomNumber,invType) do {room := roomNumber | invasionType := invType | clock := 0} lazy
[intCode = 1 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
S2_TP2
[invasionType >= 81 AND invasionType <= 280 AND invasionType = movSensorNumber AND clock <= 500] sync TriggerRoomMovementAlarm?(movSensorNumber) do {clock := 0} delayable
[invasionType >= 1 AND invasionType <= 50 AND invasionType = windowNumber AND clock <= 500] sync TriggerWindowBreakingAlarm?(windowNumber) do {clock := 0} delayable
[invasionType >= 51 AND invasionType <= 80 AND invasionType = doorNumber AND clock <= 500] sync TriggerDoorOpeningAlarm?(doorNumber) do {clock := 0} delayable
[invasionType = 0] sync NoInvasionDetected?() do {} delayable
[choice = 3 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
S3_TP2
[choice = 4 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
[room = roomNumber AND clock <= 4000] sync CallPolice?(roomNumber) do {clock := 0} delayable
[intCode = 3 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
[choice = 2 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
[choice = 5 AND interruptionClock <= 50] sync ReactivateAlarmSystem?() do {} delayable
[invasionType > 280] sync Error?() do {} delayable
S4_TP2
[intCode = 1 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
[invasionType > 280] sync Error?() do {} delayable
[invasionType = 0] sync NoInvasionDetected?() do {} delayable
[intCode = 3 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 4 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
[intCode = 5 AND choice = 0 AND NOT (intCode = 3)] sync Interrupt!(intCode) do {choice := intCode} lazy
S6_TP2
S5_TP2
[room = roomNumber AND clock <= 500] sync TurnOnLights?(roomNumber) do {} delayable
[intCode = 4 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
S6_Reject
S1_Reject
I1_Reject
[intCode = 2 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy
S5_Accept
[intCode = 5 AND choice = 0] sync Interrupt!(intCode) do {choice := intCode} lazy

Figure B.4: Synchronous Product

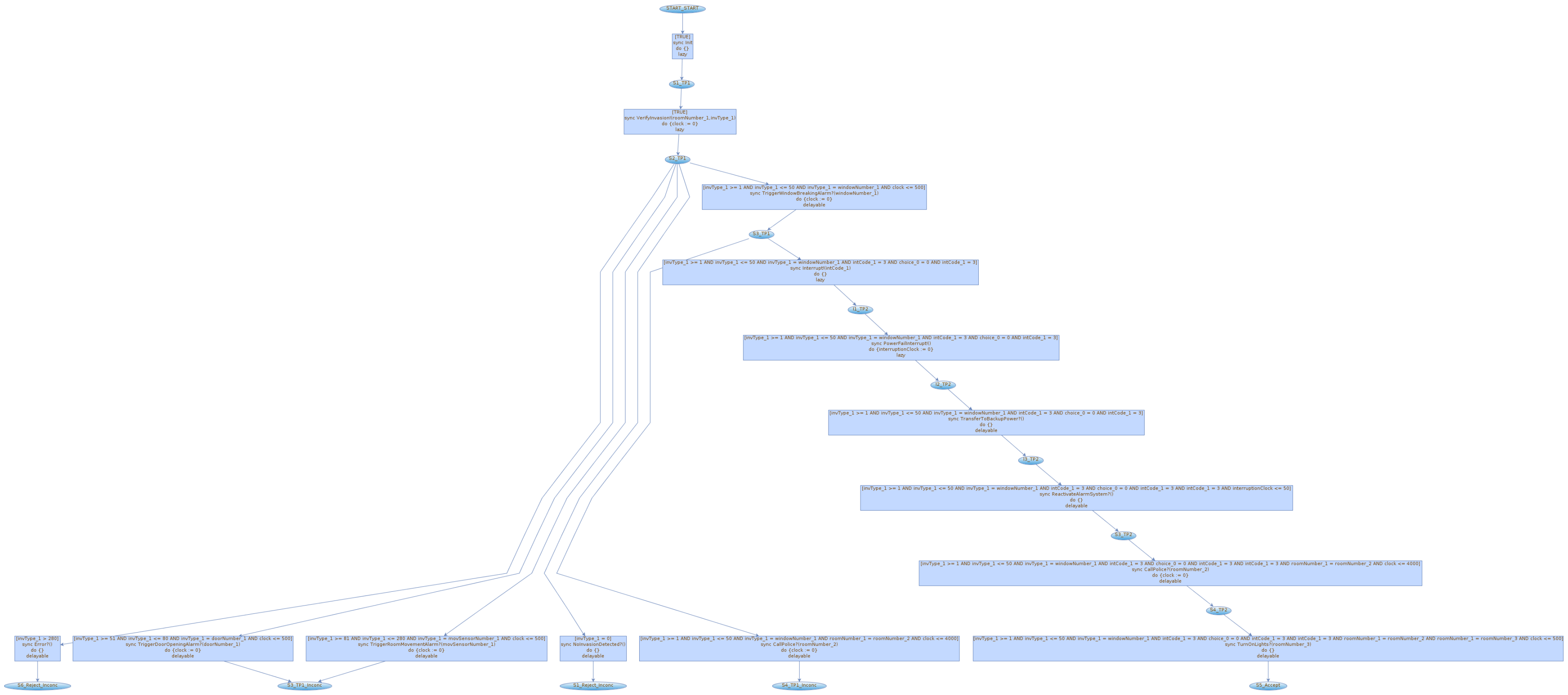

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync VerifyInvasion!(roomNumber_1,invType_1)
do {clock := 0}
lazy
S2_TP1
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND clock <= 500]
sync TriggerWindowBreakingAlarm?(windowNumber_1)
do {clock := 0}
delayable
S3_TP1
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync Interrupt!(intCode_1)
do {}
lazy
I1_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync PowerFailInterrupt!()
do {interruptionClock := 0}
lazy
I2_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync TransferToBackupPower?()
do {}
delayable
I3_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem?()
do {}
delayable
S3_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP2
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND roomNumber_1 = roomNumber_3 AND clock <= 500]
sync TurnOnLights?(roomNumber_3)
do {}
delayable
S5_Accept
[invType_1 > 280]
sync Error?()
do {}
delayable
S6_Reject_Inconc
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND clock <= 500]
sync TriggerDoorOpeningAlarm?(doorNumber_1)
do {clock := 0}
delayable
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND clock <= 500]
sync TriggerRoomMovementAlarm?(movSensorNumber_1)
do {clock := 0}
delayable
S3_TP1_Inconc
[invType_1 = 0]
sync NoInvasionDetected?()
do {}
delayable
S1_Reject_Inconc
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP1_Inconc

Figure B.5: Test Case 01

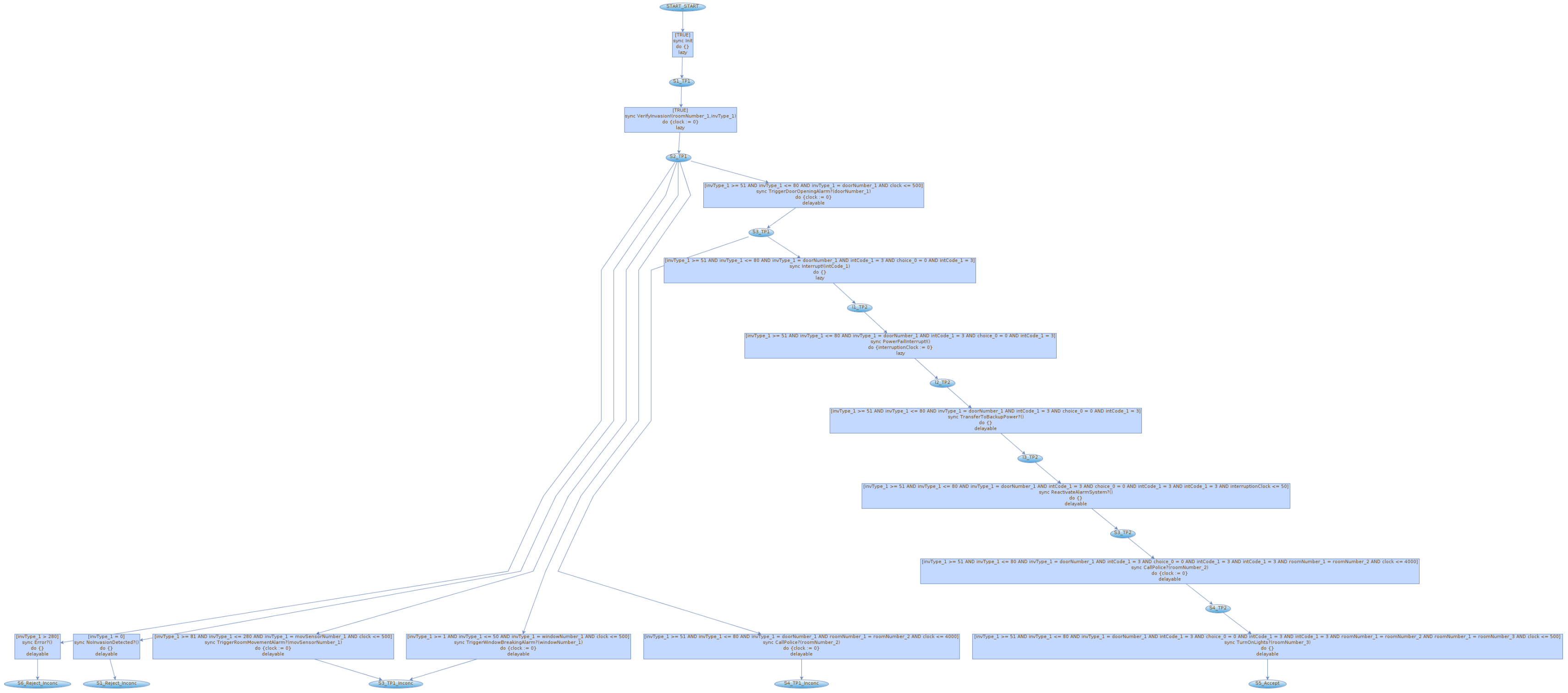

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync VerifyInvasion!(roomNumber_1,invType_1)
do {clock := 0}
lazy
S2_TP1
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND clock <= 500]
sync TriggerDoorOpeningAlarm?(doorNumber_1)
do {clock := 0}
delayable
S3_TP1
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync Interrupt!(intCode_1)
do {}
lazy
I1_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync PowerFailInterrupt!()
do {interruptionClock := 0}
lazy
I2_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync TransferToBackupPower?()
do {}
delayable
I3_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem?()
do {}
delayable
S3_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP2
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND roomNumber_1 = roomNumber_3 AND clock <= 500]
sync TurnOnLights?(roomNumber_3)
do {}
delayable
S5_Accept
[invType_1 > 280]
sync Error?()
do {}
delayable
S6_Reject_Inconc
[invType_1 = 0]
sync NoInvasionDetected?()
do {}
delayable
S1_Reject_Inconc
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND clock <= 500]
sync TriggerRoomMovementAlarm?(movSensorNumber_1)
do {clock := 0}
delayable
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND clock <= 500]
sync TriggerWindowBreakingAlarm?(windowNumber_1)
do {clock := 0}
delayable
S3_TP1_Inconc
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP1_Inconc

Figure B.6: Test Case 02

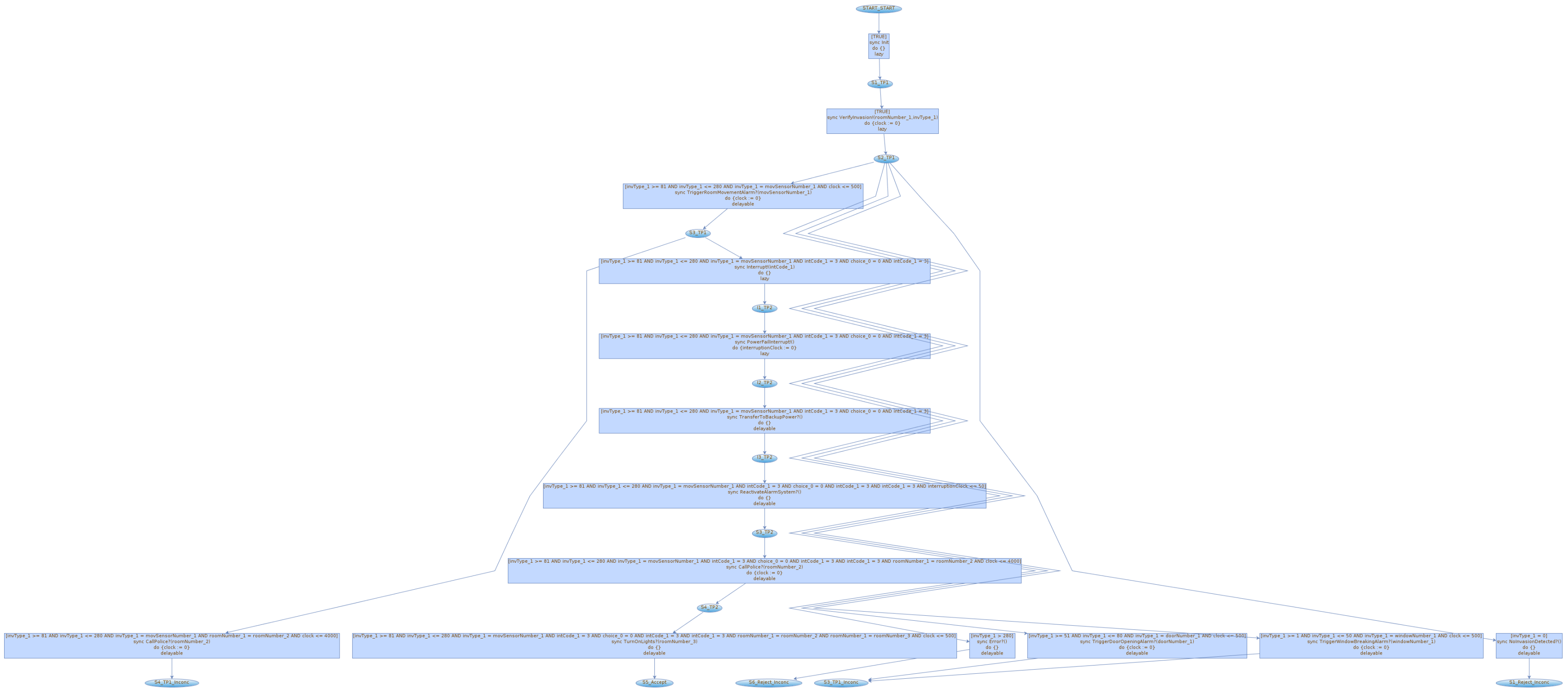

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync VerifyInvasion!(roomNumber_1,invType_1)
do {clock := 0}
lazy
S2_TP1
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND clock <= 500]
sync TriggerRoomMovementAlarm?(movSensorNumber_1)
do {clock := 0}
delayable
S3_TP1
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync Interrupt!(intCode_1)
do {}
lazy
I1_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync PowerFailInterrupt!()
do {interruptionClock := 0}
lazy
I2_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3]
sync TransferToBackupPower?()
do {}
delayable
I3_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND interruptionClock <= 50]
sync ReactivateAlarmSystem?()
do {}
delayable
S3_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP2
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND roomNumber_1 = roomNumber_2 AND clock <= 4000]
sync CallPolice?(roomNumber_2)
do {clock := 0}
delayable
S4_TP1_Inconc
[invType_1 >= 81 AND invType_1 <= 280 AND invType_1 = movSensorNumber_1 AND intCode_1 = 3 AND choice_0 = 0 AND intCode_1 = 3 AND intCode_1 = 3 AND roomNumber_1 = roomNumber_2 AND roomNumber_1 = roomNumber_3 AND clock <= 500]
sync TurnOnLights?(roomNumber_3)
do {}
delayable
S5_Accept
S6_Reject_Inconc
S3_TP1_Inconc
[invType_1 > 280]
sync Error?()
do {}
delayable
[invType_1 >= 51 AND invType_1 <= 80 AND invType_1 = doorNumber_1 AND clock <= 500]
sync TriggerDoorOpeningAlarm?(doorNumber_1)
do {clock := 0}
delayable
[invType_1 >= 1 AND invType_1 <= 50 AND invType_1 = windowNumber_1 AND clock <= 500]
sync TriggerWindowBreakingAlarm?(windowNumber_1)
do {clock := 0}
delayable
[invType_1 = 0]
sync NoInvasionDetected?()
do {}
delayable
S1_Reject_Inconc

Figure B.7: Test Case 03

B.2 TIOSTS Models of the Automatic Guided Vehicle System Case Study

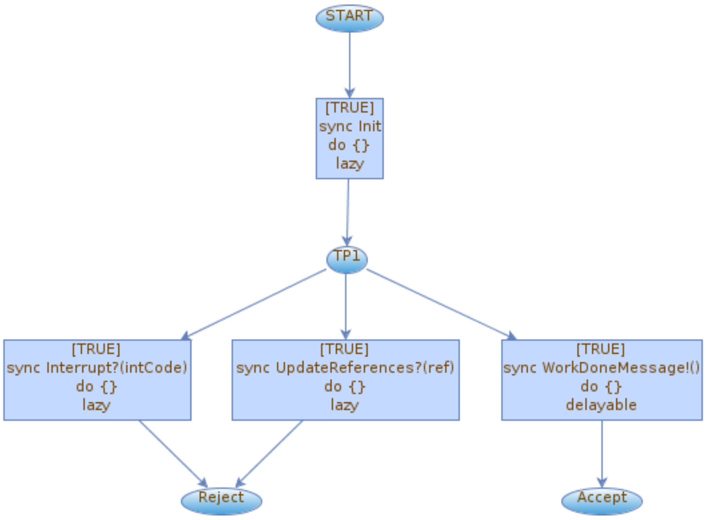


Figure B.8: Test Purpose TP1

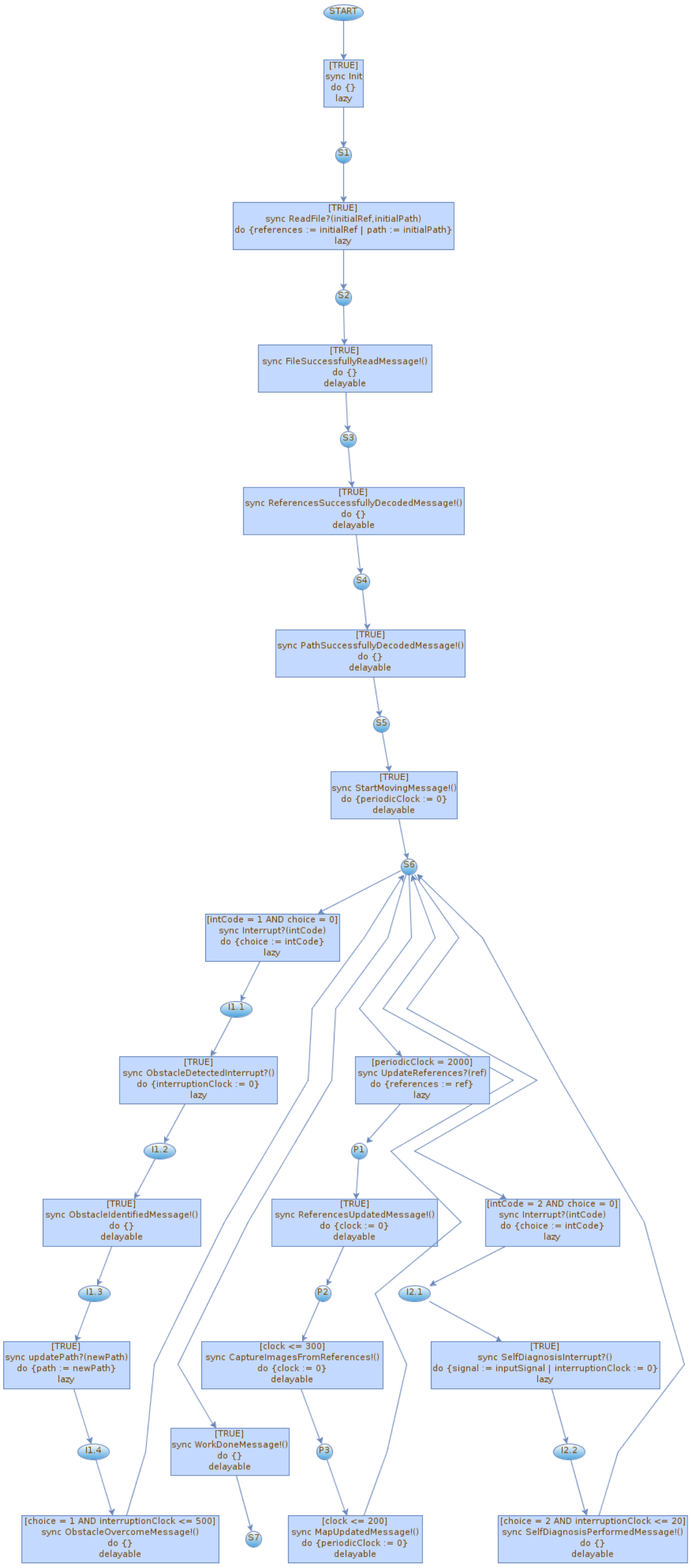

START
[TRUE]
sync Init
do {}
lazy
S1
[TRUE]
sync ReadFile?(initialRef,initialPath)
do {references := initialRef | path := initialPath}
lazy
S2
[TRUE]
sync FileSuccessfullyReadMessage!()
do {}
delayable
S3
[TRUE]
sync ReferencesSuccessfullyDecodedMessage!()
do {}
delayable
S4
[TRUE]
sync PathSuccessfullyDecodedMessage!()
do {}
delayable
S5
[TRUE]
sync StartMovingMessage!()
do {periodicClock := 0}
delayable
S6
[intCode = 1 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
I1.1
[TRUE]
sync ObstacleDetectedInterrupt?()
do {interruptionClock := 0}
lazy
I1.2
[TRUE]
sync ObstacleIdentifiedMessage!()
do {}
delayable
I1.3
[TRUE]
sync updatePath?(newPath)
do {path := newPath}
lazy
I1.4
[choice = 1 AND interruptionClock <= 500]
sync ObstacleOvercomeMessage!()
do {}
delayable
[periodicClock = 2000]
sync UpdateReferences?(ref)
do {references := ref}
lazy
P1
[TRUE]
sync ReferencesUpdatedMessage!()
do {clock := 0}
delayable
P2
[clock <= 300]
sync CaptureImagesFromReferences!()
do {clock := 0}
delayable
P3
[clock <= 200]
sync MapUpdatedMessage!()
do {periodicClock := 0}
delayable
[TRUE]
sync WorkDoneMessage!()
do {}
delayable
S7
[intCode = 2 AND choice = 0]
sync Interrupt?(intCode)
do {choice := intCode}
lazy
I2.1
[TRUE]
sync SelfDiagnosisInterrupt?()
do {signal := inputSignal | interruptionClock := 0}
lazy
I2.2
[choice = 2 AND interruptionClock <= 20]
sync SelfDiagnosisPerformedMessage!()
do {}
delayable

Figure B.9: Automatic Guided Vehicle System Specification

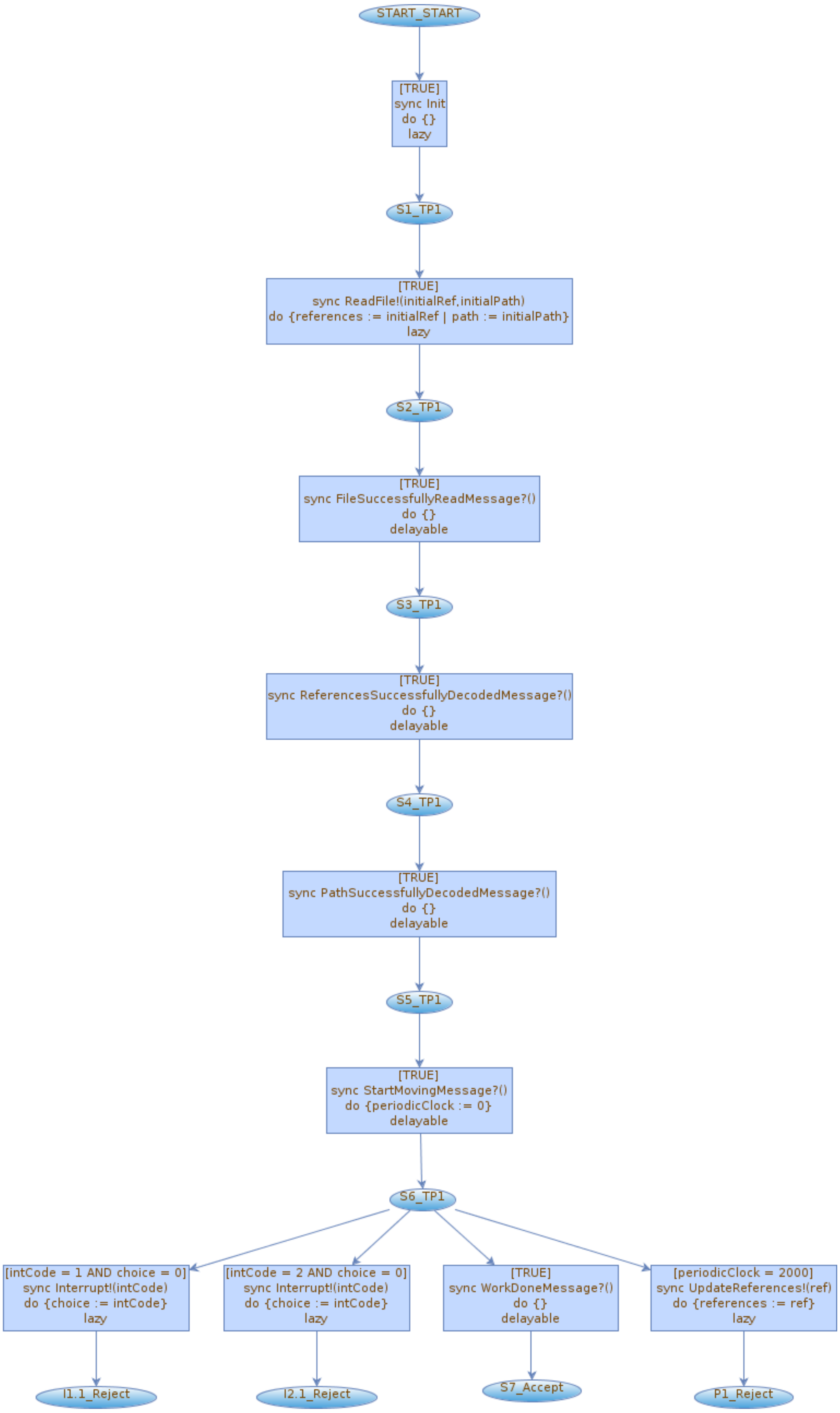


Figure B.10: Synchronous Product between AVG System Specification and Test Purpose TP1

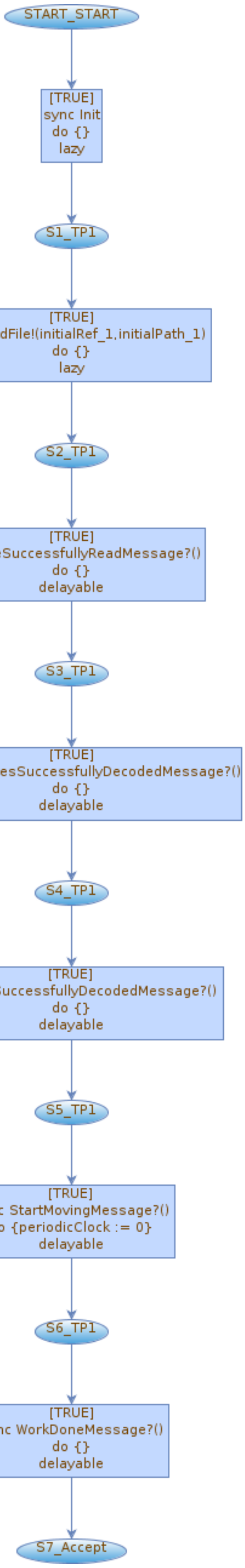


Figure B.11: Test Case of the First Scenario

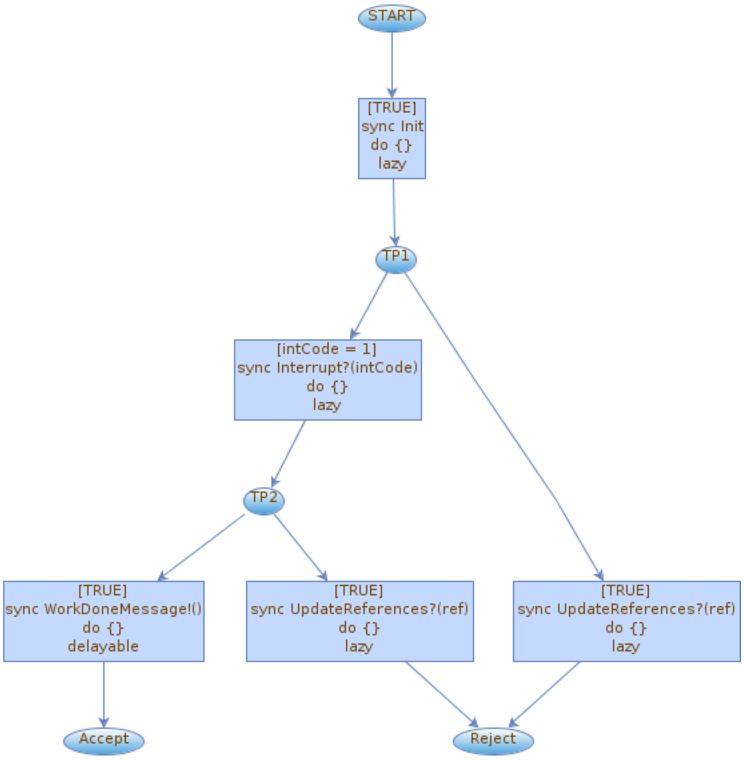


Figure B.12: Test Purpose TP2

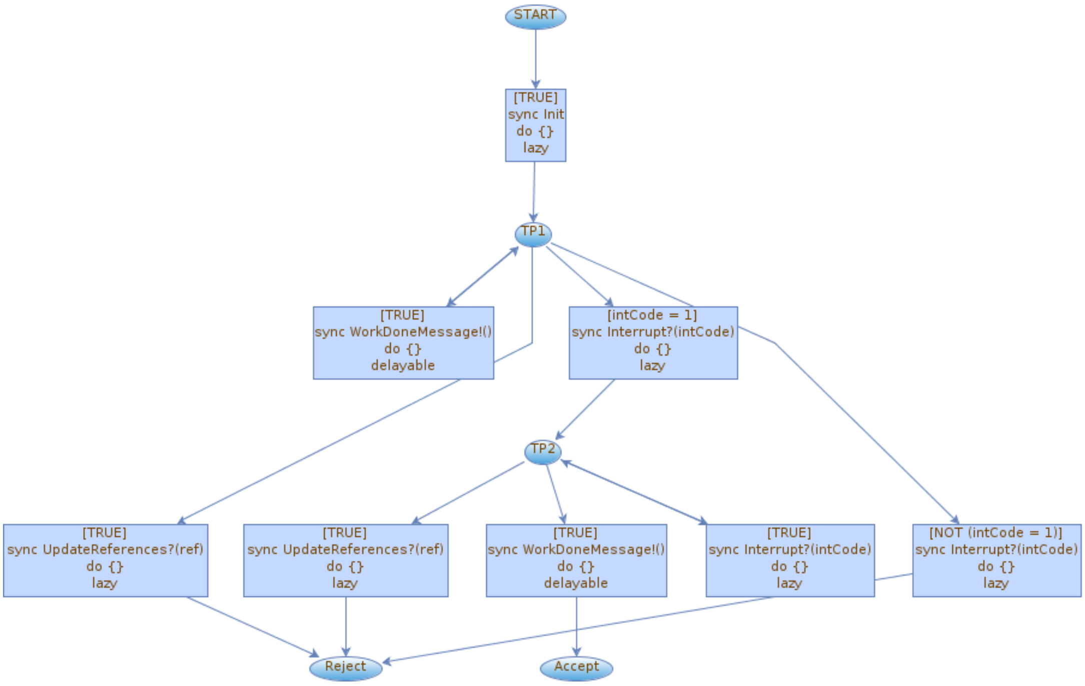


Figure B.13: Completed Test Purpose TP2

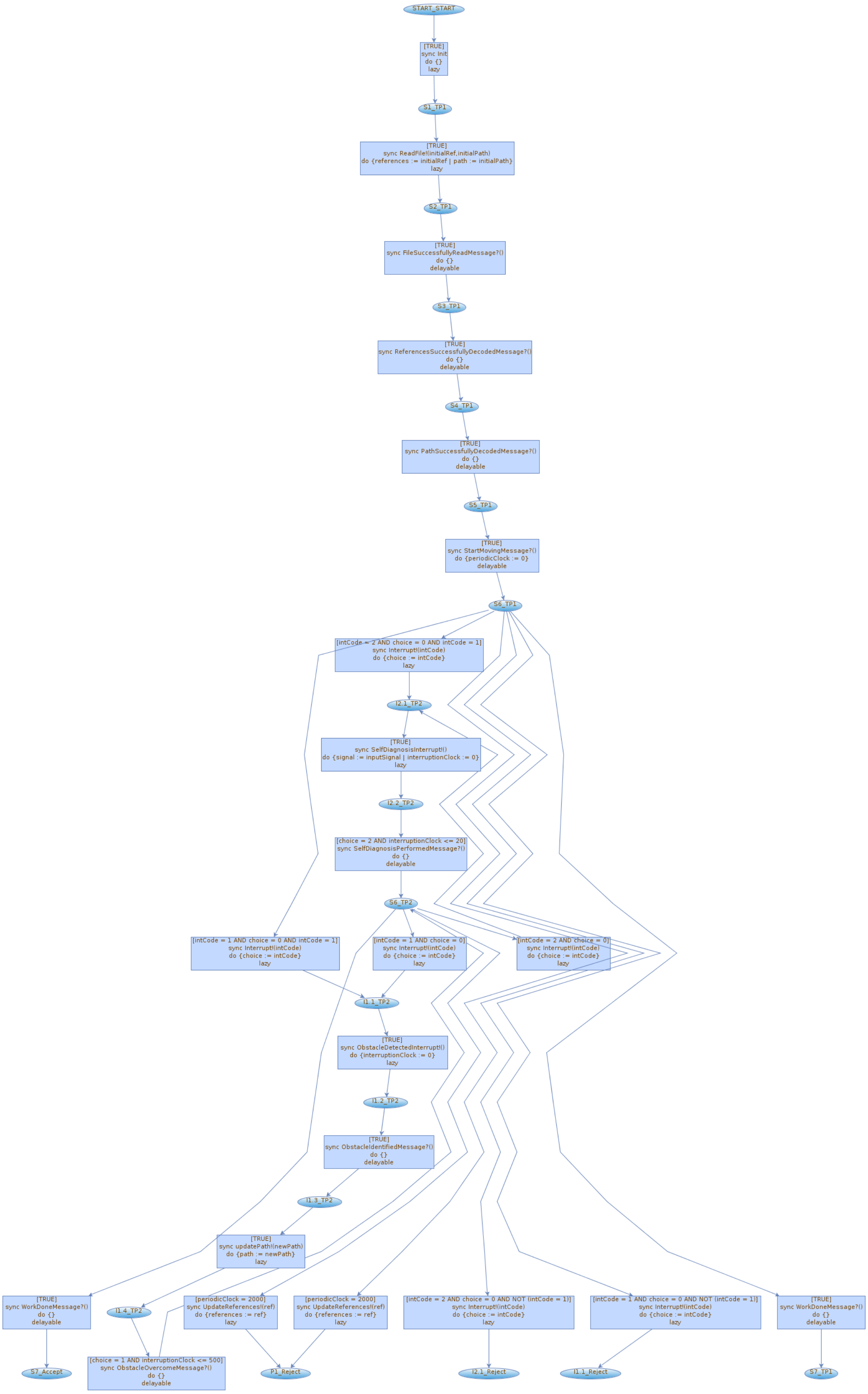

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync ReadFile!(initialRef,initialPath)
do {references := initialRef | path := initialPath}
lazy
S2_TP1
[TRUE]
sync FileSuccessfullyReadMessage?()
do {}
delayable
S3_TP1
[TRUE]
sync ReferencesSuccessfullyDecodedMessage?()
do {}
delayable
S4_TP1
[TRUE]
sync PathSuccessfullyDecodedMessage?()
do {}
delayable
S5_TP1
[TRUE]
sync StartMovingMessage?()
do {periodicClock := 0}
delayable
S6_TP1
[intCode = 2 AND choice = 0 AND intCode = 1]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
I2.1_TP2
[TRUE]
sync SelfDiagnosisInterrupt!()
do {signal := inputSignal | interruptionClock := 0}
lazy
I2.2_TP2
[choice = 2 AND interruptionClock <= 20]
sync SelfDiagnosisPerformedMessage?()
do {}
delayable
S6_TP2
[intCode = 1 AND choice = 0 AND intCode = 1]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[intCode = 1 AND choice = 0]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[intCode = 2 AND choice = 0]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
I1.1_TP2
[TRUE]
sync ObstacleDetectedInterrupt!()
do {interruptionClock := 0}
lazy
I1.2_TP2
[TRUE]
sync ObstacleIdentifiedMessage?()
do {}
delayable
I1.3_TP2
[TRUE]
sync updatePath!(newPath)
do {path := newPath}
lazy
[TRUE]
sync WorkDoneMessage?()
do {}
delayable
I1.4_TP2
[periodicClock = 2000]
sync UpdateReferences!(ref)
do {references := ref}
lazy
[periodicClock = 2000]
sync UpdateReferences!(ref)
do {references := ref}
lazy
[intCode = 2 AND choice = 0 AND NOT (intCode = 1)]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[intCode = 1 AND choice = 0 AND NOT (intCode = 1)]
sync Interrupt!(intCode)
do {choice := intCode}
lazy
[TRUE]
sync WorkDoneMessage?()
do {}
delayable
S7_Accept
[choice = 1 AND interruptionClock <= 500]
sync ObstacleOvercomeMessage?()
do {}
delayable
P1_Reject
I2.1_Reject
I1.1_Reject
S7_TP1

Figure B.14: Synchronous Product between AVG System Specification and Test Purpose TP2

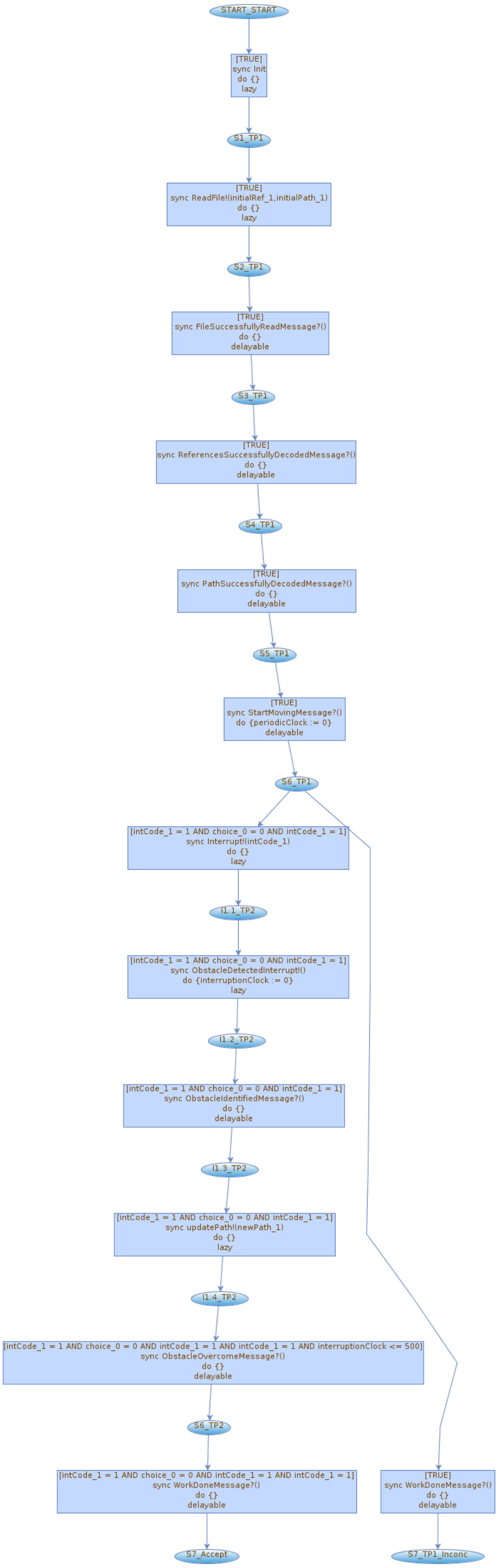

START_START
[TRUE]
sync Init
do {}
lazy
S1_TP1
[TRUE]
sync ReadFile!(initialRef_1,initialPath_1)
do {}
lazy
S2_TP1
[TRUE]
sync FileSuccessfullyReadMessage?()
do {}
delayable
S3_TP1
[TRUE]
sync ReferencesSuccessfullyDecodedMessage?()
do {}
delayable
S4_TP1
[TRUE]
sync PathSuccessfullyDecodedMessage?()
do {}
delayable
S5_TP1
[TRUE]
sync StartMovingMessage?()
do {periodicClock := 0}
delayable
S6_TP1
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync Interrupt!(intCode_1)
do {}
lazy
I1.1_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync ObstacleDetectedInterrupt!()
do {interruptionClock := 0}
lazy
I1.2_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync ObstacleIdentifiedMessage?()
do {}
delayable
I1.3_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1]
sync updatePath!(newPath_1)
do {}
lazy
I1.4_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1 AND intCode_1 = 1 AND interruptionClock <= 500]
sync ObstacleOvercomeMessage?()
do {}
delayable
S6_TP2
[intCode_1 = 1 AND choice_0 = 0 AND intCode_1 = 1 AND intCode_1 = 1]
sync WorkDoneMessage?()
do {}
delayable
S7_Accept
[TRUE]
sync WorkDoneMessage?()
do {}
delayable
S7_TP1_Inconc

Figure B.15: Test Case of the Second Scenario