UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Uma Técnica para Modelagem e Verificação de Programas Java Concorrentes Auxiliada por Anotações de Código

Elthon Allex da Silva Oliveira

Campina Grande, PB, Brasil
Junho de 2006

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Uma Técnica para Modelagem e Verificação de Programas Java Concorrentes Auxiliada por Anotações de Código

Elthon Allex da Silva Oliveira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Informática (MSc).

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

Jorge César Abrantes de Figueiredo
Dalton Dario Serey Guerrero
Orientadores

Campina Grande, PB, Brasil
Junho de 2006

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

O148t 2006

Oliveira, Elthon Allex da Silva

Uma Técnica para Modelagem e Verificação de Programas Java Concorrentes Auxiliada por Anotações de Código / Elthon Allex da Silva Oliveira - Campina Grande, 2006

148.: il.

Referências

Dissertação (Mestrado em Informática) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadores: Jorge C. A. de Figueiredo e Dalton D. S. Guerrero

1– Verificação de Software 2– Programas Concorrentes 3– Verificação de Modelos 4– Métodos Formais I-Título

CDU 004.415.22/.5+004.436

Resumo

Métodos formais vêm sendo utilizados muito hoje em dia em projetos de desenvolvimento em que há uma grande exigência de que o software se comporte exatamente conforme esperado. Entretanto, os projetos que fazem uso de métodos formais se limitam aos poucos projetos que estão dispostos a investir em recursos humanos capacitados. Neste trabalho é apresentada uma técnica desenvolvida para viabilizar a inclusão de métodos formais, mais especificamente a técnica de verificação de modelos (*model checking*), nos processos de desenvolvimento de software concorrente orientado a objetos. É então definida uma linguagem de descrição comportamental capaz de descrever e abstrair o comportamento de programas orientados a objetos com múltiplas linhas de execução. Tal linguagem, escrita junto ao código na forma de linguagem de anotação, atua como mecanismo de abstração do programa a ser verificado. Por ser uma linguagem semelhante a uma linguagem de programação, o programador é quem modela seu próprio código, dispensando o especialista que seria necessário para modelagem formal do sistema. Além disso, por ser de anotação, ameniza o problema da sincronização entre o modelo e o sistema modelado. Os modelos descritos usando a linguagem de descrição comportamental são então traduzidos para uma linguagem formal executável existente. A partir deste modelo formal e das propriedades especificadas a serem verificadas, é realizada a verificação de modelos utilizando um verificador de modelos. O desenvolvedor permanece em contato apenas com as anotações e os resultados obtidos no processo de verificação. O restante do processo ocorre de forma totalmente escondida do usuário, numa *caixa preta*.

Abstract

Formal methods have been used so much nowadays in development projects in where there is a requirement for the software behavior being as it is expected to be. However, there are few projects that are disposed to invest their money in capable human resources. In this work it is presented a technique developed for making it easier to use formal methods, model checking more precisely, in the concurrent object oriented software development processes. It is defined a behavioral description language that is able to model multi-threaded object oriented programs. Such a language, written together with the source code, in an annotation language format, acts like the abstraction of the program to be verified. Due to its similarity to a programming language, the programmer is who models its own code, not being necessary to have an expert to formal modelling of the system. Besides, also due to its annotation characteristic, it eases the synchronization problem between the model and the modelled system. The described models using the behavioral description language are translated to an existent executable formal language. With this formal model and the specified properties to be checked in hands, the model checking process is done by using a model checker. The developer just stays in direct contact with annotations and returned results from the verification process. The rest of the process occurs in a totally hidden way to the user, in a *black box*.

Agradecimentos

Há muitas pessoas a quem devo meus sinceros agradecimentos.

A Deus, aos meus pais José e Luciene, e à minha tia Irene (*in memorian*), não necessariamente nesta ordem, pela força e amor incondicionais que sempre me deram.

Aos meus orientadores, Jorge e Dalton, por terem contribuído muito na minha evolução acadêmica em todos os aspectos. Aos professores, Patrícia Machado e Angelo Percusich, por terem me dado conselhos valiosos em algumas etapas de minha vida acadêmica como mestrando, e Hyggo Almeida, por ter me dado conselhos em todas as etapas. Ao professor Evandro Costa e ao companheiro de trabalhos laterais Leandro Silva pelo apoio dado.

Ao professor Adenilso Simão e à professora Joseana Fechine pelas importantes contribuições.

À Aninha por ter sido sempre muito paciente e prestativa.

Aos amigos Loreno e Emerson Memessu por terem contribuído com os estudos de caso. Principalmente ao Loreno, que teve paciência e foi bastante prestativo com minha pessoa diversas vezes. *And to the Professor Eric Mercer, from Brigham Young University, for the help with Bogor failures.*

Aos meus grandes amigos que ganhei aqui em Campina Grande e que também foram meus "orientadores": Emerson Memessu, Glauber e Emerson Malungo. Sendo o primeiro um dos caras que me motivaram quando eu menos acreditei em mim, e o último, um irmão para mim em todos os momentos. E aos meus amigos Lauro Beltrão e Carol Medeiros pelas maravilhosas conversas.

Aos meus amigos André Atanasio, Xambinho (Alexandre), Eanes, Fred, Milena, Luana, Rômulo (o praiêro) e Wallace por terem feito parte da minha vida durante o mestrado, mesmo que geograficamente distante no caso de alguns deles. Os ótimos momentos nunca serão esquecidos.

A todos os meus amigos *xumbetas* e companheiros do LabPetri pelas discussões e brincadeiras muito valorosas: Ana Emíla, Amâncio, Cássio (meu gerente), Daniel X, Fabrício, Jaírson, João, Paulo (*ÔxeMeuAmigo*) e Taciano. Mas gostaria de agradecer principalmente aos amigos Afrânio, Daniel (Dandan Von Sta), Flávio e Rogério. Todos estes foram verdadeiros companheiros na (in)formalidade.

À Fabiana pelo apoio e pelos momentos felizes que, juntamente com a teoria da relatividade, fizeram minha estadia em Campina ter sido muito mais curta do que realmente foi.

Aos meus irmãos que deixei em Alagoas: Neymar, Adriana, Alan, Alexandre, Anna, Fernanda (minha empresária), Gustavo, Hamilton, Lívia, Paula, Plínio e Valéria. Sei que torceram por mim.

Ao CNPq e à Capes pelo apoio financeiro.

E por fim, gostaria de pedir desculpas e agradecer principalmente a todos os outros amigos, companheiros e colegas que por infelicidade eu tenha esquecido. E ao Chuck Norris, é claro!!

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos Anotados

Capítulo 1

Introdução

1.1 Contextualização

Verificação é uma atividade indispensável em qualquer processo de desenvolvimento de software. É por meio dela que são obtidas evidências da corretude do sistema desenvolvido. Apesar de sua importância, a verificação geralmente é feita de forma *ad hoc*, ou ainda pior, simplesmente inexiste nos projetos de desenvolvimento por ser subestimado o ganho obtido pelo sua prática.

As técnicas de verificação de software são usadas para detectar defeitos difíceis de serem descobertos pelo desenvolvedor [Jam88], objetivando a eliminação do maior número de defeitos possível. A utilização correta de técnicas de verificação implica na detecção destes defeitos mais difíceis de serem encontrados, pois é realizada uma análise automática e mais eficaz que a análise feita somente pelo ser humano. Implica também que, em virtude da detecção de tais defeitos, eles acabam não permanecendo na versão final do programa entregue ao cliente, evitando prejuízos. Segundo o trabalho apresentado por Stefan Wagner em [Wag05], o custo proveniente dos prejuízos causados ao cliente, pelos defeitos não detectados, somado ao custo do processo de desenvolvimento para eliminar estes defeitos, é maior do que o custo empregado para detectar e eliminar os defeitos durante a fase inicial de desenvolvimento. Com isso, o esforço empregado no uso de técnicas de verificação de software, apropriadas às necessidades do contexto em questão, é compensado pela economia de gastos com eventuais consertos de defeitos.

As técnicas de verificação de software podem ser divididas em duas classes: *verificação*

dinâmica e *verificação estática*. Na primeira, o código do programa deve ser executado para que se possa verificá-lo. Testes e avaliação de asserções no código são exemplos de verificação dinâmica. No caso das asserções, os resultados da verificação são reportados na medida em que o sistema é executado. Em muitos casos, a execução e, portanto, a verificação é feita com o sistema em produção. Desta forma, é possível que parte do comportamento indesejado seja descoberto somente quando o cliente usar o sistema. Outra desvantagem desta técnica é que o código referente à funcionalidade verificada deve estar totalmente implementado.

Na segunda classe, a de *verificação estática*, a verificação é feita sem que o código seja executado, ou seja, de forma estática. Exemplos de verificação estática são as verificações sintática e semântica e a verificação de modelos, na qual um modelo abstrato do sistema é executado. Nesta classe, o sistema não precisa estar completamente implementado, o que torna os processos de implementação e verificação mais independentes um do outro. Contudo, os modelos abstratos podem não representar corretamente o comportamento dos sistemas correspondentes face à distância sintática e semântica geralmente presente entre a linguagem de modelagem e a de programação.

Teste é uma técnica de verificação dinâmica muito conhecida e utilizada. Há vários tipos diferentes de teste de software, como testes de unidade, de regressão, de integração, entre outros [Bei90]. Os testes de unidade são os mais usados em razão da simplicidade de seu usao em relação aos demais tipos de teste. Eles tornam o processo de teste menos complexo dividindo o sistema em pequenas unidades a serem testadas isoladamente [KdB04; Sch04]. Além disso, arcabouços foram desenvolvidos para dar suporte a este tipo de teste [JUn06; PyU06; NUn06], o que torna os testes de unidade ainda mais atraentes. Um problema destes testes é que eles fornecem evidências de corretude apenas para sistemas não concorrentes. Em se tratando de sistemas concorrentes, estas evidências de corretude não são fornecidas. Isto se dá por que programas implementados usando o paradigma de programação concorrente contêm os chamados *interleavings*, ou intercalação entre as linhas de execução. Estas intercalações fazem com que os programas passem a ter um comportamento não determinístico. Em casos em que duas ou mais linhas de execução compartilham recursos, conhecidos como *race conditions* [NM92], as intercalações podem produzir diferentes resultados finais, para um mesmo dado de entrada. Em outras palavras, testes de unidade podem garantir a

corretude de sistemas concorrentes apenas para algumas execuções, mas não para todas.

Ainda no contexto de testes, há uma outra técnica de verificação. São os chamados testes formais. Neste tipo de teste, a verificação de programas concorrentes é bem mais rigorosa do que a verificação com testes *ad hoc*. A partir de uma especificação formal do sistema, os cenários de execução são definidos para que se possa construir bons objetivos de teste, e conseqüentemente, bons casos de teste para os sistemas, concorrentes ou não. Entretanto, a definição de tais cenários a serem verificados faz com que os testes não sejam suficientemente eficientes na verificação de sistemas concorrentes. Como mencionado anteriormente, os sistemas concorrentes produzem intercalações entre as linhas de execução, e isto possibilita inúmeros cenários. Desta forma, apesar de produzirem evidências de corretude do sistema verificado, os testes formais deixam de cobrir uma grande parte do comportamento do sistema.

Duas outras técnicas de verificação estática que podem ser usadas na verificação de sistemas concorrentes são os métodos dedutivos e a verificação de modelos (*model checking*). Em ambas, modelos comportamentais do sistema são construídos para a verificação. Na verificação dedutiva, as propriedades do sistema são escritas num modelo matemático a partir do qual são procuradas provas matemáticas para a corretude do sistema. Há ferramentas que dão suporte parcial a estes métodos, precisando ainda da intervenção do usuário em alguns momentos. Entretanto, métodos dedutivos são difíceis de aplicar e requerem a presença de profissionais qualificados. A prova de um simples protocolo ou circuito, que possui um comportamento bastante limitado se comparado a sistemas de software, pode durar dias ou até meses [CGP99]. Por esta razão, métodos dedutivos são pouco atrativos e difíceis de serem integrados com o desenvolvimento de software, que hoje é realizado de forma muito ágil [All01], dispensando todo artefato que não seja o próprio código fonte.

Há um trabalho em particular chamado *Perfect Developer* [Tec06] que utiliza a técnica de verificação por meio de métodos dedutivos. Neste trabalho, é usada uma linguagem formal proprietária para especificar requisitos e propriedades. Segundo a empresa proprietária da ferramenta, a partir das propriedades especificadas usando a linguagem formal, é gerado o código fonte pronto para compilação. Qualquer alteração na implementação do código fonte gerado pode tornar sua especificação obsoleta. Desta forma, para garantir a corretude do sistema pela ferramenta, qualquer evolução do sistema deve ser posto em sua especificação

e todo o código fonte deve ser gerado novamente.

A técnica de verificação de modelos, por outro lado, dispensa profissionais especialistas da mesma forma que os métodos dedutivos exigem. Um treinamento realizado para o aprendizado de uma linguagem de modelagem comportamental e de especificação de propriedades é suficiente para capacitar recursos humanos. As propriedades são expressas de forma declarativa, utilizando alguma lógica temporal, que permita expressar propriedades dos estados do sistema em função do tempo sem referência explícita a este último.

Como as técnicas clássicas, as mencionadas até o momento, apresentam alguns problemas em suas aplicações, vários trabalhos foram e vêm sendo desenvolvidos com o objetivo de possibilitar abordagens viáveis de verificação de sistemas concorrentes, usando a verificação de modelos [Kin06; SpE06; JLi06; VHBP00]. Alguns destes trabalhos extraem informações comportamentais diretamente a partir do código compilado (*bytecodes* Java) [JLi06; VHBP00], produzindo uma abstração do sistema, que é verificada de forma totalmente automática. Contudo, há problemas no trabalho apresentado por Visser [VHBP00], por exemplo, a construção do modelo comportamental é totalmente dependente do formato dos *bytecodes* gerados pelo compilador. Logo, qualquer variação no formato de *bytecodes* inviabiliza a técnica. No trabalho em que foi desenvolvida a ferramenta Jlint [JLi06], a verificação é feita sem interferência alguma do usuário. Até as propriedades verificadas pela ferramenta são pré-estabelecidas, o que facilita bastante o seu uso por usuários leigos em métodos formais. Por outro lado, esta característica representa uma grande limitação ao trabalho, pois nenhuma outra propriedade pode ser verificada junto ao software.

No trabalho apresentado por Kiniry [Kin06], informações comportamentais são extraídas diretamente a partir do código fonte (em Java) e as propriedades especificadas, manualmente pelo usuário, são verificadas. Enquanto no projeto denominado SpEx [SpE06], o código fonte (também em Java) e as propriedades, também especificadas manualmente pelo usuário, são traduzidos numa linguagem formal executável que serve como entrada para uma outra ferramenta de verificação formal. Os problemas encontrados nestes dois trabalhos são, respectivamente, a dependência de uma versão específica do compilador, que já está obsoleto, e a não abstração do código, que acarreta no problema da explosão do espaço de estados [Val98].

Há uma característica interessante em comum em alguns dos trabalhos mencionados,

[BJ01; VHBP00; SpE06; Kin06], eles usam anotações de código para instrumentar o código. Esta instrumentação tem servido como auxílio à verificação do software. Algumas anotações têm sido usadas para especificar as propriedades que se deseja verificar junto ao sistema anotado [SpE06; VHBP00; Kin06]. Elas têm sido usadas também na extensão de funcionalidades em linguagens de programação para especificação de propriedades de sistemas [LvH85], na aplicação de *design by contract* [LBR03] em programas Java, na modelagem formal da arquitetura de sistemas de software [JSHS96], e algumas destas anotações são usadas inclusive como guias aos compiladores que efetuam verificação no código [BJ01]. Para facilitar o uso de tais anotações, estes trabalhos têm desenvolvido anotações com sintaxe e semântica semelhantes as das linguagens de programação em que são anotadas.

Há duas vantagens no uso de anotações de código como instrumento de auxílio à verificação de software. Uma delas é que por estar junto ao código, é mais fácil que a informação permaneça atualizada em relação ao código fonte, assim como acontece com o uso de *Javadoc* para documentar programas Java [Kra99; PG05], pois a relação entre código e anotação se torna mais próxima. Uma outra vantagem é que além de servir como instrumento de verificação, anotações podem servir como documentação complementar do sistema, facilitando ainda mais o entendimento da equipe de desenvolvimento sobre o código. Diante deste cenário, anotações de código também poderiam ser usadas para escrever os modelos comportamentais dos programas a serem verificados. Isto possibilitaria um controle direto no nível de abstração a ser empregado no processo de verificação, o que não é feito nos trabalhos de verificação de sistemas concorrentes mencionados anteriormente.

1.2 Problema do Trabalho

Baseado na seção anterior, algumas deficiências foram apontadas no cenário de verificação de software. As técnicas de verificação atuais, que fazem uso de técnicas de verificação clássicas (e.g., *model checking*, métodos dedutivos), geralmente não dão poder ao usuário para que este possa aplicar a abstração desejada ao modelo do sistema desenvolvido. Quando dão, fornecem ao usuário, como mecanimso de abstração, alguma linguagem formal sintática e semanticamente diferente da linguagem de programação utilizada. Além disso, algumas das técnicas são dependentes de plataforma, o que limita o seu uso. A partir destas informações,

foi detectado o problema deste trabalho:

> *Definir uma técnica para modelagem e verificação de programas concorrentes que seja amigável ao usuário, leigo em métodos formais, e que possa tirar proveito das vantagens de uma técnica de verificação clássica.*

Nesta técnica, deverá ser usada uma linguagem de anotação voltada à descrição comportamental e à especificação de propriedades, que possua semântica e sintaxe parecidas com as da linguagem alvo e que contemple aspectos de concorrência. Esta linguagem de anotação deve servir como instrumentação para abstração do código fonte. Ela deve ser passível de tradução para uma linguagem de modelagem formal executável.

1.3 Solução Proposta

Neste trabalho, é apresentada uma proposta de técnica para a verificação de software concorrente auxiliada por uma linguagem de anotação voltada à descrição comportamental do código. Uma visão geral desta técnica é ilustrada na Figura 1.1. Como prova de conceitos, foi escolhida a linguagem de programação Java como sendo a linguagem alvo.

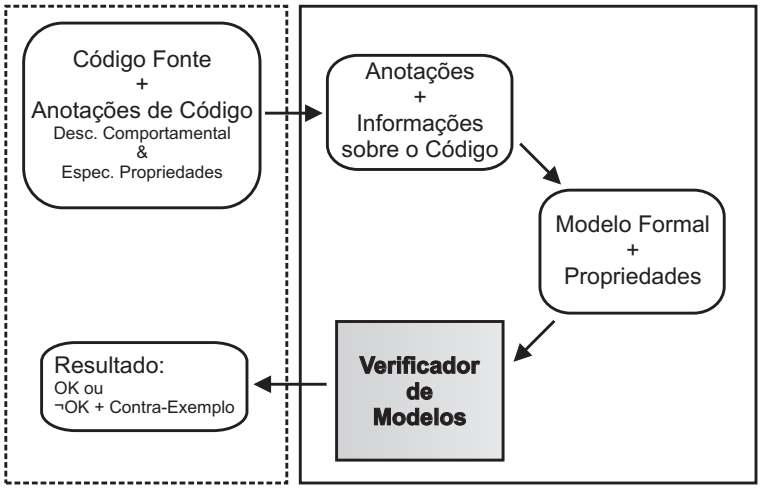

Figura 1.1: Visão geral da técnica proposta.

Dois tipos de anotação são usados pela linguagem de anotação de código aqui definida: *anotação comportamental* e *anotação de especificação*. Elas são usadas para descrever o comportamento do código do sistema e para especificar as propriedades desejadas, respectivamente. Ambas as anotações devem ser escritas pelo próprio desenvolvedor. A segunda

anotação, de especificação de propriedades, possui a semântica da lógica temporal LTL [Pnu77].

Após a fase de anotação de código, modelagem e especificação de propriedades, um processo passível de automação deve ser capaz de extrair tais anotações juntamente com algumas informações sobre o código fonte. A partir das anotações e informações sobre o código, é gerado um único artefato contendo o modelo comportamental formal do sistema e as propriedades especificadas. Este artefato único é passado a um *verificador de modelos* que verifica se o modelo gerado satisfaz as propriedades especificadas. O *verificador de modelos* é uma ferramenta usada na técnica de verificação de modelos para checar se as propriedades especificadas são satisfeitas pelo modelo formal do sistema.

Caso o modelo gerado pela ferramenta satisfaça as propriedades especificadas, uma resposta afirmativa é dada ao desenvolvedor. Caso contrário, uma seqüência de execuções, denominada *contra-exemplo*, é retornada ao usuário para que este possa detectar, nas anotações, onde está o erro. Como as anotações são inseridas junto ao código correspondente, o *possível* erro também é detectado no código fonte. É dito *possível* por que o desenvolvedor pode descrever o comportamento do sistema de forma errada. Assim, o erro pode estar no modelo descrito e não no sistema verificado. Além disso, o desenvolvedor pode também especificar a propriedade desejada de forma errada. O modelo passa então a não satisfazer uma propriedade que não deveria existir. A análise do modelo e das propriedades em busca de erros proporciona ao desenvolvedor um conhecimento mais profundo sobre seu próprio código.

Como ilustrado também na Figura 1.1, há duas demarcações feitas com caixas. A caixa ao lado esquerdo da figura indica o que deve ser visível ao usuário final: código-fonte e anotações, ambos escritos por ele; e o resultado final da verificação. Enquanto a outra caixa, do lado direito, contém o que deve permanecer escondido do usuário: processo de extração das anotações e das informações sobre o código-fonte, geração do modelo formal, e a verificação do modelo. Esta última parte é feita por uma ferramenta chamada *verificador de modelos*, já existente e desenvolvida por terceiros [Rob03].

1.4 Resultados concretos e Relevância

Algumas metas cumpridas foram a definição de uma linguagem de anotação, voltada à descrição comportamental, com sintaxe e semântica semelhantes às da linguagem de programação Java; e a implementação de um protótipo de *plugin* para o Eclipse que servisse como interface com o usuário automatizando a técnica de verificação de software aqui apresentada. Tal *plugin* é composto por alguns componentes com funcionalidades bem definidas. Dentre estes componentes, pode-se destacar: o *parser* para a linguagem de anotação definida e o tradutor desta linguagem de anotação para a linguagem formal de entrada ao verificador. Esta decomposição facilita a evolução do *plugin*. Caso a linguagem de anotação ou linguagem formal sejam alteradas, por exemplo, apenas modificações pontuais precisam der feitas. Ao fazer uso de uma mesma IDE (Eclipse) para as etapas de implementação e de verificação, estas etapas passam a ficar mais próximas uma da outra, facilitando a verificação do código fonte por parte do próprio desenvolvedor.

A concretização dos componentes mencionados (linguagem, *parser*, tradutor e *plugin*) resultou numa implementação, ou prova de conceitos, da técnica aqui definida, uma ferramenta para verificação formal de software concorrente que oferece ao usuário pleno controle sobre o nível de abstração com que se deseja trabalhar. Este controle se deve graças à linguagem de anotação definida. Linguagem esta que possui sintaxe e semântica semelhantes às da linguagem Java, o que a torna mais fácil de ser assimilada por programadores, para modelagem de sistemas de software, do que certas linguagens formais como Promela [pro06] ou Redes de Petri [Jen92], por exemplo.

Assim sendo, esta técnica possibilita ao desenvolvedor verificar formalmente o sistema em desenvolvimento, mesmo que tal sistema não esteja totalmente implementado. Por meio desta verificação, são coletadas evidências de que o sistema se comporta como esperado, passando assim mais confiança sobre ele. Contudo, uma limitação desta técnica é o uso de lógica temporal para especificar as propriedades, o que limita o seu uso completo a um pequeno grupo de programadores. Uma outra limitação diz respeito à limitação de memória inerente ao uso da técnica de verificação de modelos.

Uma constatação sobre a linguagem de anotação aqui definida foi feita. A proximidade sintática e semântica entre a linguagem de descrição comportamental e a linguagem de

programação alvo não trouxe somente benefícios, como era esperado. Foi detectado um pequeno, contudo significativo ônus. Tal proximidade faz com que o programador, inexperiente em modelagem comportamental, tenda a replicar todo o seu código.

1.5 Estrutura da Dissertação

O restante deste documento está organizado da seguinte forma:

Capítulo 2: Conceitos Fundamentais e Trabalhos Relacionados Neste Capítulo são apresentados alguns trabalhos desenvolvidos com o propósito de serem usados na verificação de software. Suas características principais são mencionadas, ressaltando suas limitações.

Capítulo 3: Linguagem JaCA A linguagem de descrição comportamental definida neste trabalho é apresentada neste Capítulo. Sintaxe e semântica da linguagem são mostradas através de pequenos exemplos.

Capítulo 4: O Plugin Neste Capítulo é apresentada a implementação da solução mostrada neste capítulo introdutório, uma ferramenta no formato de um *plugin* desenvolvido para ser usado juntamente com a *IDE* de desenvolvimento de software Eclipse. É feita uma apresentação de alto nível sobre como usar a ferramenta e quais suas características principais. Outra apresentação é dada, agora em baixo nível, sobre o funcionamento dos componentes e como se dá a interação entre eles.

Capítulo 5: Experimentação e Validação Neste Capítulo é mostrada a verificação de dois sistemas. Num primeiro momento, são validadas a técnica e a ferramenta desenvolvidas. O sistema a ser verificado e seu comportamento são apresentados. Em seguida são enumeradas algumas propriedades desejadas que foram verificadas junto ao modelo comportamental. Algumas anotações comportamentais e todas as anotações de especificação de propriedades são mostradas. Estas anotações são explicadas de forma breve. Os resultados obtidos a partir dos experimentos e seus dados são mostrados. Num segundo momento, é validada a linguagem de anotação aqui proposta. Por meio de uma experiência com um desenvolvedor

leigo em modelagem formal, alguns resultados são obtidos. E por fim, as conclusões sobre todo o processo referente aos dois estudos de caso concluem este capítulo.

Capítulo 6: Considerações Finais Por último, são apresentadas as conclusões sobre o trabalho, enfatizando os resultados obtidos, contribuições feitas e alguns possíveis trabalhos futuros.

Apêndice A: Verificação de Modelos Neste apêndice é feita uma apresentação sobre a técnica de *model checking*. São mostradas as duas lógicas temporais usadas para descrição de propriedades e uma breve comparação entre o poder de expressividade das duas é feita.

Apêndice B: O Verificador *Bogor* e a Linguagem *BIR* Neste apêndice são apresentados o verificador de modelos Bogor, de forma breve, e algumas estruturas de sua linguagem de modelagem, BIR.

Apêndice C: Equivalências entre *JaCA* e *BIR* Neste apêndice são apresentadas as equivalências entre as estruturas da linguagem de anotação definida neste trabalho e da linguagem de modelagem BIR.

Apêndice D: Experiência com o Desenvolvedor As informações obtidas a partir da experiência com um programador são apresentadas neste apêndice.

Apêndice E: Código e anotações dos sistemas estudados Este apêndice contém todo o código fonte verificado nos dois estudos de caso e suas respectivas anotações JaCA.

Apêndice F: Instalação e uso do *plugin* Este apêndice contém as instruções de como instalar o protótipo do *plugin* implementado.

Capítulo 2

Conceitos Fundamentais e Trabalhos Relacionados

Neste capítulo é apresentada brevemente a técnica de verificação de modelos, utilizada neste trabalho, e alguns conceitos relacionados a tal técnica. Em seguida, é apresentada uma breve introdução sobre os requisitos de uma técnica, ou metodologia, para verificação formal de software, e os requisitos da linguagem de descrição comportamental a ser usada. Logo em seguida, algumas linguagens de modelagem comportamental e de especificação de propriedades são discutidas. Algumas delas são linguagens de anotação. Também são discutidas as ferramentas de verificação que fazem uso destas linguagens e outras ferramentas que extraem o modelo comportamental diretamente a partir do código. Além disso, é apresentada uma abordagem de geração de modelo comportamental do projeto Spex [SpE06].

2.1 Conceitos Fundamentais

A verificação de modelos (*model checking*) é uma técnica automática para analisar o espaço de estados finito de sistemas concorrentes [CWA+96]. Na Figura 2.1, é mostrado o esquema desta técnica. Sua aplicação ocorre tradicionalmente através de três etapas:

1. Modelagem: esta etapa consiste em construir um modelo formal do sistema, fazendo uso de alguma linguagem formal, e a partir dele, obter *todo* o comportamento possível do sistema. O modelo que contém todo o comportamento obtido a partir do modelo formal é conhecido como *espaço de estados* do sistema.

2. Especificação: esta etapa consiste em especificar as propriedades comportamentais *desejáveis* do sistema. Um comportamento que se deseja do sistema pode ser descrito formalmente através de lógicas temporais ou máquinas de estado.

3. Verificação: esta etapa consiste em verificar se as especificações escritas são satisfeitas pelo modelo (espaço de estados). Esta etapa é feita de forma automática pela ferramenta chamada *verificador de modelos*. Esta ferramenta produz como resultado um valor verdade que indica se a especificação é satisfeita ou não pelo modelo. Em caso negativo, o verificador retorna ao usuário uma seqüência de estados de uma determinada execução, chamada de *contra-exemplo*, que demonstra que a especificação não é válida no modelo.

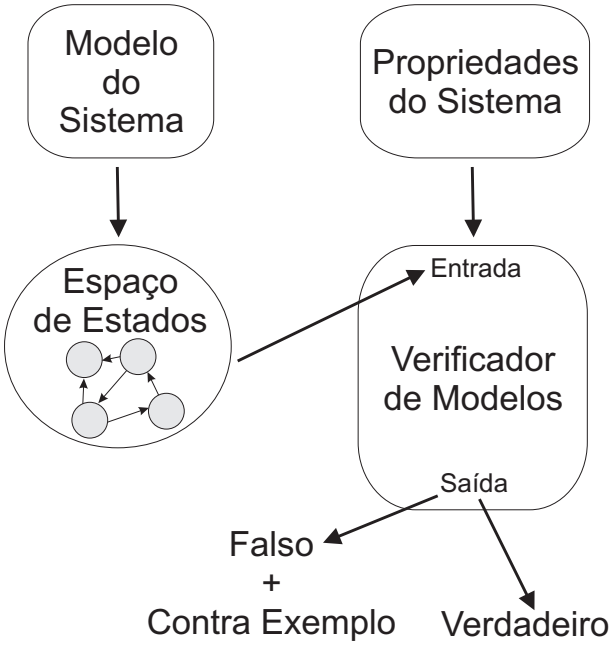

Figura 2.1: Esquema da técnica de verificação de modelos.

As lógicas temporais são utilizadas para a especificação de propriedades em verificação de modelos porque são capazes de expressar relações de ordem, sem recorrer à noção explícita de tempo. Tradicionalmente, duas formalizações de lógica temporal são utilizadas no contexto de verificação de modelos: LTL (*Linear Temporal Logic*) [Pnu77] e CTL (*Computation Tree Logic*) [Sif90]. A abordagem em LTL, conhecida também como lógica temporal linear, considera que uma propriedade pode ser quantificada para todas as execuções do sistema. A abordagem em CTL, também conhecida como lógica temporal ramificada, por

sua vez, considera que uma propriedade pode ser quantificada para uma ou para todas as execuções do sistema. No trabalho aqui apresentado, é utilizada a lógica temporal LTL em virtude do verificador de modelos utilizado, o verificador Bogor [Rob03].

O principal desafio à aplicação de verificação de modelos em situações reais é o famoso problema conhecido como *explosão do espaço de estados*. Armazenar todos os comportamentos possíveis de um sistema, mesmo sendo um sistema simples, pode esgotar os recursos de memória de uma máquina, mesmo que o número de estados alcançados pelo sistema seja finito. Este espaço de estados é representado por uma *estrutura de Kripke*.

Uma *estrutura de Kripke* é uma máquina de estados finita que representa todas as possíveis execuções de um sistema, ou como na maioria das vezes, todas as possíveis execuções do modelo de um sistema. Cada estado do sistema é rotulado com as proposições atômicas que são verdadeiras nele. Cada estado nesta estrutura deve possuir ao menos um sucessor. Situações reais nas quais um estado s não possui um sucessor (*deadlock*) são representadas através de auto-laço, sucessor do estado s é o próprio estado. Na Figura 2.2, é apresentado um exemplo desta estrutura para um sistema que utiliza quatro processadores. O estado denotado por $S_{0,0}$ indica que o sistema não está em funcionamento. Já os estados $S_{n,1}$ representam o funcionamento do sistema com *n+1* processadores funcionando.

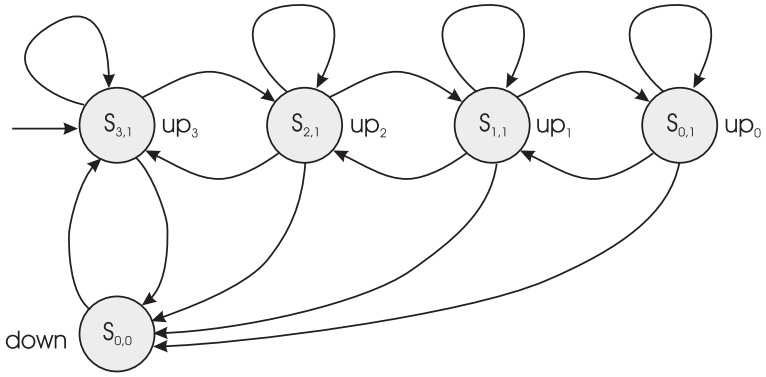

Figura 2.2: Exemplo de estrutura de Kripke.

A partir de uma *estrutura de Kripke* são obtidos os *caminhos* nos quais as propriedades LTL serão avaliadas. Um *caminho* é uma seqüência infinita de estados que representa uma possível execução do sistema a partir do seu estado inicial. Fórmulas LTL são avaliadas em todos os caminhos possíveis de uma determinada *estrutura de Kripke*. Desta forma, determinada fórmula LTL é válida no modelo do sistema definido se e somente se ela é

válida para todos os caminhos do modelo que iniciam em algum estado inicial. Alguns caminhos para a estrutura apresentada na Figura 2.2 são: up_3, up_2, up_3, up_2, ...; up_3, up_2, up_3, *down*, up_3, ...; up_3, up_2, up_1, up_0, *down*, up_3, ...; etc.

As fórmulas em LTL são capazes de expressar relações de ordem, sem recorrer à noção explícita de tempo. Um exemplo de propriedade descrita em fórmula LTL, para a estrutura de Kripke apresentada na Figura 2.2, é: <> *down*. Esta fórmula significa que "*futuramente o sistema não estará funcionando*". Para tal estrutura de Kripke esta fórmula se mostra falsa. Um possível contra-exemplo, dentre inúmeros, é o caminho em que o estado up_3 sempre se repete.

No Apêndice A são apresentados formalmente todos os conceitos contidos nesta seção.

2.2 Trabalhos Relacionados

2.2.1 Introdução

Alguns fatores importantes devem ser levados em consideração antes de se usar uma linguagem de descrição comportamental e de especificação de propriedades. O fator considerado menos importante neste trabalho diz respeito a qual linguagem deve ser linguagem alvo. O que deve ser considerado importante neste aspecto é que, independente de linguagem alvo, a linguagem de anotação deva ser sintática e semanticamente parecida.

Dois fatores importantes a serem considerados são: a capacidade da linguagem, que irá auxiliar a verificação, de modelar o código e especificar suas propriedades; e a semelhança sintática e semântica entre esta linguagem e a linguagem de programação modelada. Como é desejado que o próprio desenvolvedor modele seu código, quanto maior forem estas semelhanças, menor será o esforço empregado pelo desenvolvedor para usar a linguagem de descrição. Além disso, é desejado que o desenvolvedor tenha controle absoluto sobre o nível de abstração empregado no processo de verificação.

Um outro fator é o suporte à modelagem e especificação de propriedades no contexto de programas concorrentes. O não suporte à concorrência significa deixar de fora algumas aplicações desenvolvidas hoje em dia. E são justamente estas aplicações, nas quais erros são muito difíceis de serem detectados, que mais carecem de verificação.

E por último, e não menos importante, a linguagem em questão deve ser de anotação. Pois, como mencionado anteriormente, estudos mostram que o simples fato de o modelo permanecer junto ao código fonte, formando um único artefato, diminui bastante o problema da sincronização entre o código fonte e a sua descrição [Kra99; PG05].

2.2.2 Linguagens de anotação

ANNA - Annotaded Ada

Ada é uma linguagem de programação criada pelo departamento de defesa dos Estados Unidos utilizada em serviços de telecomunicações, militares e aeroespacial[1]. Para a verificação dos programas implementados em Ada, foi desenvolvida a linguagem ANNA (*ANNotated Ada*) [LvH85].

ANNA é uma linguagem de programação de alto nível que estende Ada com vários tipos de construtores de especificação. ANNA especifica as propriedades desejadas e seu compilador cria asserções para serem verificadas junto ao código em tempo de execução. Esta linguagem inclui construtores de anotação especiais, ou asserções de semântica, que estabelecem axiomas sobre as *procedures*. Tal linguagem é baseada na lógica de primeira ordem e inclui também restrições de tipo generalizadas e construtores de especificação comportamental, que vão de simples asserções até especificações algébricas complexas.

Além disso, muitos dos conceitos de Ada foram aproveitados para a linguagem ANNA. Em muitos casos, outras características foram acrescentadas aos conceitos. Como por exemplo, as expressões booleanas em Ada foram incrementadas com quantificadores, tornando-se assim as expressões booleanas de ANNA. Uma outra característica da linguagem de anotação ANNA, é a separação das anotações em *contextos*. Cada pedaço de anotação é válido apenas em seu contexto, e este é definido pelo próprio especificador.

Contudo, ANNA apresenta algumas limitações. O próprio compilador, que verifica as propriedades diretamente junto ao código fonte, não permite que o desenvolvedor interfira no nível de abstração a ser empregado. Além disso, a verificação de propriedades é feita em tempo de execução do programa. Outra desvantagem de ANNA é o não suporte a programas

[1] O foguete espacial *Ariane 5* que explodiu pouco depois da decolagem tinha seu sistema programado em Ada.

concorrentes.

JML - Java Modeling Language

JML (*Java Modeling Language*) [LBR03] é uma linguagem de especificação comportamental de interfaces voltada para a linguagem Java. Esta linguagem suporta quantificadores, variáveis de especificação e outros recursos que a tornam bastante expressiva.

JML descreve o comportamento dos módulos de um sistema (classes) do ponto de vista do cliente. Ela é usada para especificar as interfaces dos módulos, ou seja, seus nomes e tipos de campos e métodos usando uma sintaxe semelhante à de Java, seguindo a mesma idéia da linguagem de especificação ANNA.

As informações das especificações comportamentais aparecem na forma de anotações. A linguagem proporciona ao usuário um conjunto de classes puras[2] incluindo conjuntos, seqüências, relações, *maps*, entre outras, que são úteis em especificações comportamentais. Estas classes podem ser usadas em outras especificações como modelos conceituais em *designs* detalhados. O usuário (programador geralmente) pode criar suas próprias classes puras para especificações mais específicas.

JML possibilita o uso de lógica temporal, mais especificamente CTL [Sif90], na verificação modular do código. Uma característica importante desta linguagem, é que ela permite que uma porção de código ainda não implementada seja especificada. Além disso, JML possibilita a expressão de invariantes e a aplicação de *design by contract* por meio de pré e pós-condições. Estes recursos fazem com que o programador passe a pensar mais no *design* do programa.

JML é uma linguagem muito expressiva. Porém, apesar de possuir construções para suportar verificação modular de programa concorrentes [RDF+05], ainda não há ferramentas que suportem tais construções. Além disso, ela possui algumas limitações para anotar o código: ela é uma linguagem de descrição comportamental de interface, e não de modelagem de comportamento dos métodos propriamente dito. Desta forma, não é possível traduzir anotações JML em modelos comportamentais. Esta linguagem de anotação tem sido

[2] A avaliação de um atributo por meio de uma classe (ou método) pode alterar o estado do objeto corrente, ou o estado de um de seus componentes. Neste caso, a classe (ou método) é causadora de mutação ou de efeitos colaterais. Uma classe (ou método) que não provoca mutação, nem efeitos colaterais, é chamada(o) de *pura(o)*.

usada em dois trabalhos no contexto de verificação de programas concorrentes [RDH04; FLL+02]. No primeiro, as anotações são traduzidas para asserções que são verificadas junto ao modelo comportamental gerado diretamente a partir do código fonte. No segundo trabalho, as anotações são traduzidas para condições de verificação a serem usadas em provadores automáticos de teoremas.

AAL - Alloy Annotation Language

A linguagem AAL (*Alloy Annotation Language*) [KMJ02] assemelha-se à linguagem JML em seu objetivo de prover uma abordagem leve para anotação de código. AAL faz uso da sintaxe de JML para definir pré e pós-valores de estado. Ela se baseia numa lógica de primeira ordem simples com operadores relacionais.

AAL permite que especificações parciais de métodos sejam escritas. Estas especificações podem ser executadas como asserções que são avaliadas em tempo de execução. Esta linguagem suporta análise automática em tempo de compilação, que pode ser usada para verificar o código contra a especificação. Esta verificação pode produzir contra exemplos, os quais podem ser usados para gerar casos de teste de forma completamente automática (através de invariantes e pré- e pós-condições).

A análise do código é feita em tempo de compilação. Desta forma, não é necessário que todo o código do programa tenha sido implementado. Com isso, o programador é incentivado a adotar o estilo de programação *test-first*, fazendo com que ele se preocupe mais com o comportamento do sistema, assim como em JML. Além disso, toda a especificação AAL é automaticamente traduzida, juntamente com o programa Java, para Alloy [Jac02] e é verificada com o *Analisador Alloy* (AA).

AAL não suporta programas com características de concorrência e paralelismo [RRDH04]. O usuário não possui qualquer controle sobre o nível de abstração empregado. Além disso, ainda não há uma ferramenta que possa traduzir as especificações e o código Java para a linguagem Alloy, os trabalhos realizados são traduzidos de forma totalmente manual.

TROLL - Textual Representation of an Object Logic Language

TROLL (*Textual Representation of an Object Logic Language*) [JSHS96] é uma linguagem utilizada na modelagem formal de sistemas de informação reativos orientados a objetos. Esta linguagem foi desenvolvida para ser usada tanto na fase de modelagem conceitual quanto na fase de especificação de requisitos do processo de desenvolvimento, e é independente da linguagem de programação.

Dentre suas características, destacam-se: é orientada a objetos, tem suporte a concorrência, é abstrata, e sua modelagem é baseada no ciclo de vida dos objetos. Esta linguagem tem uma semântica formal e pode ser traduzida para uma lógica temporal. Desta forma, as especificações feitas possuem a vantagem de terem um significado preciso e sem ambigüidades, o que torna o modelo passível de verificação. Este tipo de linguagem serve como uma ferramenta para especificar propriedades e evolução de objetos.

Existe uma versão gráfica para esta linguagem, chamada OMTROLL. Juntas, ambas propiciam uma especificação híbrida, combinando notação gráfica (OMTROLL) e textual (TROLL). A notação gráfica serve para descrever estruturas globais e suas correlações e a notação textual é usada para descrever propriedades locais e para detalhar descrição de comportamento, restrições e relacionamentos.

A linguagem TROLL apresenta uma limitação a ser levada em consideração: segundo seus desenvolvedores, ela foi projetada para ser utilizada nas fases de análise e concepção do projeto. Deste modo, ela não foi projetada para ser usada durante o desenvolvimento do sistema, mas sim apenas na fase inicial do projeto. Ou seja, TROLL não está preparada para eventuais mudanças no decorrer de um projeto. Tais mudanças tornam o modelo formal do sistema obsoleto. Outra desvantagem é que a ferramenta que suporta OMTROLL e TROLL não consegue manter os próprios modelos gráfico e textual sincronizados.

Perfect Language

Perfect Language [Lim06] é uma linguagem textual utilizada na ferramenta *Perfect Developer*, muito parecida com uma linguagem de programação. Esta linguagem é utilizada para descrever o modelo e os requisitos do usuário. Tais requisitos são inseridos dentro do modelo, ou seja, modelo e requisitos formam um único artefato.

Destinada à modelagem de aplicações orientadas a objeto críticas, esta linguagem é de fácil entendimento e engloba tanto a especificação formal dos requisitos (em forma de contratos), como a especificação do corpo de cada método. Nesta linguagem de especificação, diferentemente das outras, o modelo especificado serve como artefato para geração automática do código fonte. Desta forma, o tempo gasto com a construção do modelo pode ser compensado pela redução do tempo empregado em codificação. Esta é uma boa característica, pois diminui um pouco os esforços dos programadores na fase de implementação, já que os esforços foram empregados na fase de especificação. Uma outra característica da linguagem é que a especificação OO das entidades é feita separadamente da descrição de suas implementações.

Como limitação, destaca-se o fato de que esta linguagem e seu suporte ferramental não suportam nenhum tipo de concorrência. Não é possível verificar propriedades de sistemas concorrentes com esta linguagem [CMM05]. Além disso, as especificações de propriedades não ficam junto ao código fonte. E a qualquer alteração no modelo formal do sistema ou no código fonte gerado, o código fonte deve ser gerado novamente para que sua corretude seja garantida pela especificação.

BSL - Bandera Specification Language

A *Bandera Specification Language* [ROB00], ou simplesmente BSL, é uma linguagem para anotação de código desenvolvida para ser utilizada pela ferramenta Bandera. Bandera é uma ferramenta que oferece suporte à geração de modelos comportamentais e verificação diretamente a partir do código fonte. Esta ferramenta faz uso da técnica de $model\ checking$ (verificação de modelos) para verificar os programas. Os modelos, traduzidos diretamente a partir do código para uma linguagem intermediária, são convertidos para linguagens de entrada de verificadores de modelos, como SMV [McM00], SPIN [Hol97], JPF [VHBP00], etc.

Para a definição de propriedades a serem verificadas, o Bandera utiliza padrões de especificação [DAC99] e a linguagem BSL. Esta linguagem de especificação permite definir asserções e predicados que podem ser anotados no código do programa, como comentários ao estilo Javadoc. A partir das anotações em BSL, as asserções e os predicados, que são utilizados pelos padrões para especificação de propriedades temporais, são verificados dire-

tamente pelo conjunto de ferramentas Bandera [DHJ+01a].

Esta linguagem é dividida em duas partes: uma para asserções e outra para definição de propriedades temporais. A sub-linguagem de asserções permite que desenvolvedores definam *constraints* que podem ser verificadas no programa. É possível definir asserções a serem verificadas junto ao código fonte. Estas asserções podem ser definidas no formato de pré- e pós-condições a serem avaliadas em determinados locais específicos do código. Para que não seja necessário excluir determinada asserção, é possível nomeá-la para que ela seja desconsiderada do processo de verificação. Desta forma, podem-se montar esquemas de verificações específicas para cada situação desejada. Ou até mesmo, pode-se manter um histórico de asserções.

Como a linguagem é utilizada apenas para a especificação de propriedades, o modelo comportamental do sistema é extraído diretamente a partir do código Java. Devido a complexidade dos programas (presença de muitas variáveis e muitas linhas de execução), o processo de extração enfrenta o problema da explosão do espaço de estados. Para isso, a ferramenta que usa esta linguagem implementa alguns algoritmos de abstração. Estes algoritmos podem não ser eficazes em todos os casos, o que representa uma limitação importante. Não há controle sobre o processo de abstração do programa verificado.

Comparação entre as linguagens

Todas as linguagens apresentadas neste capítulo possuem características interessantes e desempenham muito bem as suas funções em seus respectivos contextos. Contudo, cada uma delas apresenta pelo menos uma limitação. Na Tabela 2.1é fornecida uma lista com as linguagens apresentadas neste capítulo. Para cada uma destas linguagens, são mostrados quais dos requisitos mencionados na Seção 2.2.1 são suportados. Pode-se ver que nenhuma delas suporta todos os requisitos. A linguagem BSL é a que mais se aproxima, faltando apenas poder descrever o modelo abstrato do código.

Característica / Linguagem	ANNA	JML	AAL	TROLL	Perf. Lang.	BSL
Linguagem alvo	Ada	Java	Java	OO	C++/Java/Ada	Java
Assemelha-se a ling. alvo	√	√	√			√
Descrição comportamental				√	√	
Suporta concorrência		√		√		√
Linguagem de anotação	√	√	√			√

Tabela 2.1: Tabela comparativa entre linguagens de especificação e modelagem.

2.2.3 Ferramentas e projetos de verificação

Esc/Java2

Esc/Java2 [Kin06] é uma ferramenta para verificação de programas Java. Ela faz uma análise estática sobre o código em busca de erros comuns de tempo de execução. Esta ferramenta usa um provador automático de teoremas, chamado Simplify[3] [DNS03], para verificar as propriedades especificadas.

A ferramenta confronta as decisões de *design* formalmente anotadas no código Java com a linguagem JML. Além de verificar as inconsistências entre *design* e código, também fornece advertências sobre potenciais erros presentes na implementação, que podem gerar falhas em tempo de execução. Os erros que podem ser detectados são: referência a ponteiros nulos, erros de indexação de *arrays*, erros de *cast* de tipos, dentre outros.

Uma característica desta ferramenta é o uso de uma abordagem de verificação modular, que consiste em operar sobre cada método e classe especificados separadamente. Com isso, não é necessário ter todo o código do programa disponível para usar a ferramenta.

Apesar de estar em constante evolução, a ferramenta Esc/Java2 foi desenvolvida única e exclusivamente para um certo formato dos arquivos Java compilados. Com a atualização dos compiladores Java, esta ferramenta passa a não suportar os novos *bytecodes* gerados. Poderia ser argumentado que o suporte se limita apenas a novos recursos. Contudo, caso um projeto seja compilado por uma versão mais nova do compilador Java, mesmo que use apenas os recursos da versão suportada, a ferramenta Esc/Java2 não será capaz de verificá-lo.

[3]Este provador automático de teoremas está, atualmente, sendo substituído por uma nova geração de provadores chamada SMT-LIB [RT05].

Perfect Developer

O *Perfect Developer* [Tec06] é uma ferramenta comercial, desenvolvida pela empresa Eschertech. A idéia principal sobre a qual a ferramenta foi desenvolvida é que código final da aplicação deve ser uma implementação de sua especificação correspondente.

Com o uso de um provador automático de teoremas, a ferramenta é capaz de provar a conformidade entre a especificação e os requisitos denotados nessa linguagem. A partir daí, a ferramenta Perfect pode gerar o código Java, Ada ou C++ funcional que satisfaça as especificações. Caso o código gerado precise de modificações para melhorar o desempenho, por exemplo, é sugerido que o modelo abstrato seja atualizado e que o código seja novamente gerado (refinamento da especificação ao código fonte).

Uma outra funcionalidade desta ferramenta é a possibilidade de importar modelos UML, como diagrama de classes, para a linguagem formal utilizada. Com isso, o *designer* poderia usar UML, que é uma linguagem de modelagem padrão no desenvolvimento de software, para modelar a arquitetura do sistema. Assim, este modelo seria importado para a ferramenta e posteriormente seriam descritos cada classe e método do modelo através da adição das devidas especificações (na linguagem formal) às classes obtidas pela importação.

Segundo a empresa desenvolvedora da ferramenta, algumas constatações foram feitas sobre ela:

- O código gerado é em média duas vezes maior (em linhas de código) do que a especificação.
- A menor porcentagem das *condições de verificação* que foram provadas automaticamente foi de 96%.
- O maior projeto estudado foi o da própria ferramenta, em que foram geradas 230.000 linhas de código C++, com cerca de 13.000 condições de verificação. Para provar tais condições foram necessárias cerca de 18 horas.

Em relação ao último item, pode-se constatar que o tempo necessário para provar as condições de verificação foi curto, se comparado ao tempo gasto antigamente com provadores de teoremas, cerca de dias ou até meses [CGP99]. Contudo há algumas implicações no uso desta ferramenta para o desenvolvimento de sistemas formalmente verificados. Primeiro,

cada um dos desenvolvedores envolvidos no processo deve conhecer a linguagem usada pela ferramenta, a *Perfect Language*. Assim, a reposição de algum programador da equipe de um determinado projeto implica no treinamento do novo integrante para a capacitação na linguagem usada. Segundo, dada à evolução dos sistemas, cada *condição de verificação* alterada implica numa nova geração de código fonte, que demanda tempo, resultando num grande tempo total gasto no final do processo de desenvolvimento. Além disso, qualquer alteração feita diretamente no código fonte pode comprometer a corretude garantida pela ferramenta *Perfect Developer*.

Uma desvantagem desta ferramenta é que, como mencionado anteriormente, ela e sua linguagem formal não suportam nenhum tipo de concorrência. Não é possível verificar propriedades de sistemas concorrentes. Além disso, a ferramenta não é forte o suficiente para provar técnicas de refinamento sem um trabalho adicional excessivo [CMM05].

Jlint

O Jlint [JLi06] é uma ferramenta voltada à verificação estática de programas Java. Ela é distribuída juntamente com os fontes sem quaisquer restrições. Esta ferramenta foi desenvolvida por Konstantin Knizhnik na Moscow State University e foi estendida por Cyrille Artho para suportar mais verificações de sincronização, na Swiss Federal Institute of Technology (ETHZ).

Esta ferramenta encontra possíveis erros, problemas de inconsistência e sincronização em programas Java. Para isso, ele faz análise de fluxo de execução e constrói um grafo fechado referente ao programa. O Jlint consiste de dois programas, um que analisa a semântica (problemas relacionados à sincronização, herança e fluxo de dados) e outro que analisa a sintaxe (AntiC). O programa chamado AntiC verifica alguns problemas sintáticos herdados da linguagem C.

O verificador semântico extrai informação das classes Java. Através da análise de invocações globais de métodos é possível detectar a passagem de possíveis parâmetros nulos. Além disso, a partir das dependências entre as classes, Jlint constrói um grafo que é usado para detectar situações que podem causar *deadlock* (apenas em nível de métodos em que o *synchronized* é aplicado) em sistemas concorrentes. Apesar de não conseguir detectar todos os problemas encontrados em sistemas *multithread*, Jlint pode apontar problemas que

consumiriam muito tempo para serem manualmente encontrados, como por exemplo *race conditions*.

Esta ferramenta lê os *bytecodes* para extrair as informações. A partir deles, ela busca por erros pré-estabelecidos em sua implementação. Assim sendo, uma limitação desta ferramenta é a falta de uma linguagem de anotação de código, ou qualquer outro mecanismo, que modele o comportamento do sistema ou que especifique as propriedades a serem verificadas. A verificação da ferramenta fica limitada às propriedades pré-estabelecidas.

Java PathFinder

Java PathFinder (ou simplesmente JPF) [VHBP00] é uma máquina virtual Java que executa o programa não somente uma vez como as máquinas virtuais normais, mas teoricamente, executa o programa de todas as formas possíveis, verificando violações de propriedades como *deadlocks* ou exceções não tratadas ao longo de todos os caminhos de execução em potencial. A ferramenta conta com várias técnicas de abstração para amenizar o problema da explosão do espaço de estados [JPF05].

Esta ferramenta recebe como entrada os *bytecodes* executáveis de programas Java e retorna um relatório com avisos sobre possíveis erros do sistema. Para tornar a verificação de modelos, diretamente a partir do programa, praticável, JPF emprega heurísticas e abstrações de estados no processo de verificação.

A ferramenta é capaz de verificar se as propriedades especificadas pelo usuário são satisfeitas pelo programa. Como a técnica de verificação de modelos é feita diretamente a partir de abstrações automáticas do código, não há um modelo comportamental do sistema descrito pelo usuário. O usuário deve apenas implementar as propriedades (asserções) em classes e métodos. Estes métodos devem ser chamados em determinados pontos do código do programa onde se deseja que as propriedades sejam verdadeiras.

Assim como a ferramenta Esc/Java2, apresentada na Seção 2.2.3, a ferramenta JPF não lê *bytecodes* gerados por um compilador Java diferente o compilador para o qual foi projetado. Assim, nenhum programa implementado com novos recursos pode ser verificado. Também não podem ser verificados programas que não usem estes novos recursos, mas que foram compilados por qualquer outro compilador Java. Como mencionado anteriormente, o formato dos *bytecodes* gerados difere entre as versões dos compiladores.

Projeto Spex

Spex [SpE06] é um projeto de pesquisa do laboratório SAnToS (*Specification, Analysis, and Transformation of Software*), da Universidade de Kansas, o mesmo laboratório do projeto Bandera. Ainda não há ferramenta alguma disponível, no entanto vale a pena comentar a abordagem que está sendo atualmente estudada pelo grupo para a verificação automática de programas Java anotados.

Esta abordagem visa verificar especificações, anotadas junto ao código usando JML, através da técnica de verificação de modelos. Para isto é utilizado um arcabouço de verificador de modelos desenvolvido pelo próprio grupo chamado Bogor, apresentado no Apêndice B.

A idéia deste projeto é traduzir o código JML juntamente com o programa Java num modelo comportamental escrito na linguagem de entrada do verificador Bogor, a linguagem BIR, também apresentada no Apêndice B. De posse do modelo, gerado a partir do código e das propriedades descritas geradas a partir de anotações em JML, é executada a verificação de modelos propriamente dita.

Atualmente o projeto tem estendido o verificador de modelos Bogor e a linguagem BIR para que seja possível suportar todas as construções da linguagem de especificação JML. Também não há ferramenta alguma disponível que faça a tradução direta de Java para BIR.

A conversão direta do código Java para o modelo em BIR, sem intervenção do usuário, é bastante atraente. Esta conversão fazia parte dos pontos a serem incluídos no trabalho apresentado neste documento. Contudo, alguns experimentos realizados mostraram que mesmo em programas relativamente simples, a tradução direta gera modelos comportamentais não suficientemente abstratos. Algumas traduções manuais foram feitas e foi detectado que o problema da explosão do espaço de estados é iminente, mesmo em programas não muito complexos [OM06]. Esta abordagem fica então limitada a verificar pequenas porções de código, não levando em consideração a comunicação com outras partes do sistema do qual faz parte.

Comparação entre os trabalhos

Alguns requisitos foram considerados essenciais para que uma ferramenta possa suprir a carência, no que diz respeito à verificação, dos projetos que desenvolvem sistemas reativos

não críticos, contudo ainda sim complexos. A Tabela 2.2 relaciona os trabalhos apresentados até o momento com tais requisitos.

Característica / Trabalho	ESC/Java2	JPF	Epex	Jlint	Perf. Develop.
Definição de propriedades	●	●	●		●
Abstração desejada					
Integrado a um ambiente de desenvolvimento		○			
Suporte à concorrência	●	●	●	●	
Suporte à criação dinâmica de objetos	●	●	●		
Sincronização mais eficiente entre modelo e código	●	●	○		

Tabela 2.2: Tabela comparativa entre os trabalhos apresentados.

Dentre os trabalhos, o que mais se aproxima de atender aos requisitos é a ferramenta *Java PathFinder*. Contudo, além de não ser possível definir o nível de abstração a ser usado, tal ferramenta apresenta uma interface não muito simples. Faz-se necessário importar alguns de seus arquivos para o projeto a ser verificado e as propriedades devem ser implementadas em métodos e estes devem ser invocados em determinados pontos do código fonte.

2.3 Conclusões

Neste capítulo foram apresentados os conceitos de verificação de modelos (*model checking*), estrutura de *Kripke*, e lógica temporal LTL, envolvidos neste trabalho. Também foram discutidas as linguagens de modelagem comportamental e de especificação de propriedades. Foram definidos os requisitos que uma linguagem deveria ter para ser usada no contexto do trabalho aqui apresentado.

Como apresentado, tais requisitos não foram encontrados em nenhuma das linguagens pesquisadas. Em virtude disto, fez-se necessária a definição de uma nova linguagem para fazer parte deste trabalho.

As ferramentas de verificação existentes também foram discutidas. Algumas delas não

suportam verificação de programas concorrentes [Tec06], ou não permitem que o usuário especifique suas próprias propriedades [JLi06]. Há também outras ferramentas que são dependentes de plataformas (formato dos *bytecodes*) [Kin06; VHBP00]. Além disso, há outro trabalho [SpE06] que não aplica abstração alguma na construção do modelo, o que fatalmente leva ao problema da explosão do espaço de estados.

Capítulo 3

A linguagem *JaCA*

Neste capítulo é apresentada a linguagem de descrição comportamental definida neste trabalho, que serve como linguagem de modelagem a ser anotada junto ao código fonte. Chamada de *Java Code Annotation*, ou simplesmente *JaCA*, esta linguagem é bastante semelhante à linguagem de programação orientada a objetos Java. Por último, são apresentadas algumas conclusões sobre a definição e uso desta linguagem de descrição comportamental.

3.1 Introdução

Diante das limitações presentes nas linguagens apresentadas no Capítulo 2, alguns requisitos foram definidos como sendo essenciais para uma linguagem de descrição comportamental ser usada por programadores leigos em métodos formais. Por leigos em métodos formais, entenda-se todo programador que não tenha conhecimento em linguagem de modelagem formal alguma. Os requisitos mencionados são:

- deve ser semelhante a uma linguagem de programação, fazendo com que o programador se sinta confortável em modelar seu programa, e para que seja mais fácil refletir o comportamento do programa em seu modelo;
- deve modelar características de concorrência, não se restringindo apenas a aplicações com uma única linha de execução;
- e deve ser de anotação, diminuindo assim o problema de sincronização entre os artefatos: modelo e código modelado.

A semelhança mencionada no primeiro ítem não se refere somente à parte sintática da linguagem, mas também à parte semântica. Uma grande dificuldade na modelagem de sistemas se dá devido à lacuna existente entre a semântica da linguagem de modelagem e a semântica da linguagem de programação. Devido a esta lacuna, parte do comportamento presente no modelo é modelada utilizando outros conceitos geralmente muito diferentes daqueles usados nas linguagens de programação. Isto faz com que haja, por exemplo, certa dificuldade em se manter uma fidelidade comportamental do modelo em relação ao programa correspondente.

Além da semelhança com a linguagem de programação, outro fator deve ser considerado. Restringir a linguagem de descrição comportamental a apenas programas com uma única linha de execução, ou *single thread*, significa deixar de lado uma grande parcela dos sistemas implementados hoje em dia. E é nesta parcela que estão os sistemas que mais precisam de uma verificação mais rigorosa devido à imprevisibilidade de estados causada pela intercalação (*interleaving*) das linhas de execução. Intercalação esta que, como mencionado anteriormente, torna o uso de testes de unidade e revisões em pares pouco efetivos.

E por último, como mencionado no Capítulo 1, estudos mostram que o simples fato das anotações estarem juntas com o código fonte faz com que a sincronização entre eles seja maior do que se o conteúdo das anotações estivesse presente num artefato separado do código [Kra99; PG05].

3.2 *JaCA*

A linguagem JaCA foi definida baseando-se em dois aspectos: a sintaxe e a semântica da linguagem Java e alguns recursos providos pela linguagem de modelagem BIR. Ou seja, alguns comandos e estruturas foram herdados sintática e semanticamente da linguagem de programação Java e da linguagem de modelagem BIR. Outras partes sofreram algumas pequenas alterações para que certos aspectos fossem possíveis de serem modelados.

Como um dos objetivos na definição desta linguagem é que ela seja assimilada o mais facilmente possível pelo programador, as suas estruturas são bastante similares ou iguais às estruturas da linguagem Java, como criação de objetos, por exemplo. Entretanto, de forma alguma é desejado fornecer ao programador mais uma linguagem de programação. Assim sendo, a linguagem não possui todos os recursos que Java possui. Não há uma *api* com

um conjunto de classes implementadas que possa ser usada pelo programador, como há em alguns trabalhos que usam a linguagem JML.

Deve ficar claro que JaCA é uma linguagem de modelagem, e não de programação. Há um certo limite sobre os recursos da linguagem. Um exemplo de limitação dos recursos é a manipulação de elementos de uma coleção, que em JaCA, é sempre tratado como *array* e seu limite deve ser expresso em sua instanciação. O acesso a elementos não se dá por meio de chamadas de métodos, em vez disso é usado o índice: *alunos[3]*, indicando o quarto aluno do *array*. A linguagem de anotação aqui apresentada não suporta conjuntos, tabelas *hash* ou *maps*.

As anotações de JaCA no código podem ser feitas por meio de quatro *tags*: *@att*, *@start*, *@behavior* e *@property*. As anotações feitas ao nível de classe, anotadas com *@att*, contêm as declarações de atributos que a classe deve possuir no modelo comportamental do programa a ser verificado. Os tipos suportados são: *int*, *float*, *byte*, *String*, *boolean* e *array*. Caracteres são modelados por *String* com um único caracter e qualquer identificador diferente destes mencionados é considerado classe. O nome das classes pode ser composto por letras e números em qualquer ordem. Contudo deve obrigatoriamente começar com letra.

Um exemplo de anotação ao nível de classe pode ser visto no Código Anotado 3.1. Uma classe $Estudante$ possui cinco atributos, porém seu modelo comportamental conterá apenas três, pois apenas os atributos id, $nome$ e $aprovado$ foram declarados na anotação. O contexto das anotações nas *tags* é delimitado pelo uso de parênteses, como pode ser visto na *tag @att* apresentada no Código Anotado 3.1.

```
@att(
  int id;
  String nome;
  boolean aprovado;
)
public class Estudante {
  private int id;
  private String nome;
  private String nomeDoPai;
  private String nomeDaMae;
  private int anoDeNascimento;
  private boolean aprovado;
  ...
}
```

Código Anotado 3.1: Exemplo de anotação ao nível de classe.

Um outro aspecto a ser mencionado é que, se tratando de modelo comportamental JaCA, não há diferença entre o que é atributo publico e o que é atributo privado. O que deve ser explicitado nas anotações é se o atributo em questão é ou não estático. Caso o atributo $aprovado$ da classe $Estudante$ fosse estático, bastaria ao programador adicionar a palavra chave $static$ antes da declaração do tipo da variável na anotação feita, resultando em $static\ boolean\ aprovado = false;$.

Análoga à anotação apresentada no Código Anotado 3.1, a descrição dos atributos na linguagem TROLL [JSHS96] é apresentada na Figura 3.1. TROLL foi a linguagem que mais se assemelhou à linguagem de anotação JaCA, no que diz respeito às características de suporte à concorrência e à descrição comportamental (Tabela 3.1).

```
data types nat, bool, string;
attributes
  id:nat;
  nome:string;
  aprovado:bool;
```

Figura 3.1: Exemplo de modelagem de atributos usando TROLL.

As outras três *tags* mencionadas anteriormente são anotadas diretamente juntas aos métodos. Elas representam três tipos de anotação: de início, comportamental e de especificação de propriedades. A anotação de início é definida pela *tag @start*. Este tipo de anotação não contém informação adicional como as demais *tags*. Ela serve apenas para indicar qual método deve ser o ponto inicial de execução do modelo a ser gerado. Geralmente o método a ser anotado com esta cláusula é um método $main$. Contudo, nada impede que o programador queira verificar apenas parte de seu sistema modelado anotando outro método qualquer como ponto inicial de execução. O uso desta cláusula é bastante simples e um exemplo pode ser visto no Código Anotado 3.2.

```
@start
public static void main(String[] args) {
  ...
}
```

Código Anotado 3.2: Exemplo de anotação ao nível de método usando *@start*.

A *tag @behavior* contém as anotações, ao nível de método, que expressam o comportamento, ou a abstração do comportamento, do método anotado. O comportamento é definido

pela linguagem JaCA, cuja sintaxe e semântica são apresentadas mais a frente. A modelagem (abstração) pode ser feita em sistemas implementados com uma linguagem orientada a objetos semelhante a Java e a C++, pois a linguagem de anotação aqui apresentada pode modelar objetos e múltiplas linhas de execução.

No Código Anotado 3.3, é mostrada a modelagem de criação de um objeto. Pode-se perceber que não há diferença sintática alguma entre o modelo e o código. Esta semelhança diminui o ônus inerente ao aprendizado de uma nova linguagem.

```
@behavior(
  ...
  Estudante estudante = new Estudante();
  ...
)
public static void main(String[] args) {
  ...
  Estudante estudante = new Estudante();
  ...
}
```

Código Anotado 3.3: Exemplo de anotação para criação de objetos.

Assim como na criação de objetos, a invocação de métodos feita na linguagem de anotação JaCA também não difere da sintaxe do código fonte Java. No Código Anotado 3.4, pode-se ver um exemplo de modelagem de invocação de método.

```
@behavior(
  ...
  estudante.getName();
  ...
)
public static void main(String[] args) {
  ...
  estudante.getName();
  ...
}
```

Código Anotado 3.4: Exemplo de anotação para invocação de método.

Na modelagem de laços, seja um laço implementado com $do...while$ ou com for, usa-se a estrutura *loop(<condição>){ ... }*. O que estiver compreendido entre as chaves será executado até que a expressão booleana representada por *<condição>* seja avaliada para falso. No Código Anotado 3.5, há uma modelagem de um laço implementado usando a estrutura de repetição for.

```
@behavior(
  ...
  int i = 0;
  loop(i<estudante.notas.length){
    ...
    i++;
  }
)
public float somarNotas(Estudante estudante){
  float result = 0;
  for (int i = 0; i < estudante.notas.length; i++) {
    result = result + estudante.notas[i];
  }
  return result;
}
```

Código Anotado 3.5: Exemplo de anotação usando $loop$.

Uma característica peculiar da estrutura $loop$ é que é possível colocar uma expressão matemática que resulte em algum número inteiro no lugar da expressão booleana. Desta forma, caso a expressão tenha como resultado um determinado valor n, o laço será executado n vezes. De posse deste recurso, uma anotação equivalente e mais simples do que a anotação mostrada no Código Anotado 3.5 é possível de ser feita. Esta anotação é mostrada no Código Anotado 3.6.

```
@behavior(
  ...
  loop(this.notas.length){
    ...
  }
)
public float somarNotas(Estudante estudante){
  ...
}
```

Código Anotado 3.6: Exemplo de anotação equivalente usando $loop$.

Caso seja necessário modelar um laço infinito, deve-se colocar $true$ como <*condição*>. Além disso, como forma de modelar certas abstrações do código sem que algum possível fluxo de execução do programa seja omitido na modelagem, o programador pode mesclar expressões booleanas e/ou expressões matemáticas que resultem em números inteiros. Estas expressões podem ser postas, separadas pelo símbolo '|', como <*condição*>. No Código Anotado 3.7, é mostrada a modelagem de um laço usando este recurso. Nesta modelagem,

as variáveis que determinavam o número de iterações do laço foram abstraídas. Contudo, três diferentes fluxos de execução estão modelados: a) o laço não é executado vez alguma, graças à cláusula $false$; b) o laço é executado infinitas vezes em virtude da cláusula $true$; e c) o laço é executado oito vezes. Estes fluxos de execução diferentes são refletidos na geração do espaço de estados que conterá todo o comportamento possível do modelo do sistema.

```
@behavior(
  ...
  loop(false|true|8){
    ...
  }
)
public float somarNotas(Estudante estudante){
  ...
}
```

Código Anotado 3.7: Exemplo de anotação usando $loop$ com fluxos de execução diferentes.

Esta flexibilidade da estrutura $loop$ não parece ser muito relevante neste exemplo simplório. Porém, em laços mais complexos, em que há invocações de métodos, criações de linhas de execução e manipulação de variáveis que podem ou não determinar o número de execuções do laço, esta abstração se revela bastante útil no que diz respeito à redução do espaço de estados gerado a partir do modelo. Pois, as condições e suas variáveis podem ser excluídas sem que deixe ser explorado fluxo de execução algum do programa.

Na Figura 3.2 é apresentado a estrutura para laços da linguagem TROLL. Faz-se necessário usar conjuntos para executar as iterações do laço. Em JaCA o mapeamento entre o laço em Java e o laço no modelo é feito de forma mais direta.

```
foreach <conjunto> do
  ...
od
```

Figura 3.2: Exemplo de modelagem de laço usando TROLL.

Para a inicialização de linhas de execução (ou *threads*), usa-se a estrutura $start\{$ *...* $\}$. Entre as chaves é colocado o objeto, ou os objetos caso haja mais de um, que contém o código que será executado na linha de execução a ser iniciada. No Código Anotado 3.8, é mostrada a inicialização de duas linhas de execução. Dois objetos que estendem a classe *Thread* foram criados e instanciados. Depois, seus métodos $start()$ são invocados.

```
@behavior(
  ...
  start{
    threadA;
    threadB;
  }
)
public static void main(String[] args) {
  ...
  threadA.start();
  threadB.start();
  ...
}
```

Código Anotado 3.8: Exemplo de anotação usando $start$ para criação de *threads*.

Para a modelagem de desvios condicionais, seja usando $if...then...else...$ ou $switch$, usa-se a estrutura $choice$*{ <condição1>: { ... } <condição2>: { ... } <condiçãoN>: { ... } else { ... } }*. Ela funciona da seguinte forma: ao ser executado o bloco $choice$, todas as condições avaliadas em $true$ terão seus blocos de instruções executados. Blocos estes, que estão contidos entre as chaves presentes logo após as respectivas condições. Assim como na estrutura $loop$ com múltiplas opções de condição, para cada condição verdadeira, é construída uma linha de execução no espaço de estados gerado[1]. Caso nenhuma das condições seja avaliada em $true$, o bloco de instruções referente ao $else$ é então executado [2]. No Código Anotado 3.9, é mostrada a modelagem de uma estrutura de if's.

Como forma de abstrair certas variáveis presentes nas condições da estrutura $choice$, pode-se colocar $true$ no lugar de cada um dos termos representados por <*condição*>. Desta forma, nenhum dos diferentes fluxos de execução deixará de ser coberto no modelo comportamental do programa devido à abstração aplicada. No Código Anotado 3.10, é mostrado uma estrutura condicional em que todas as condições foram abstraídas. A variável $media$ não mais existe no modelo. Assim sendo, todas as condições foram trocadas por $true$, a cláusula $else$ também foi substituída por $true$. Isso se deve por causa da semântica da estrutura $choice$. Como mencionado anteriormente, o bloco de instruções referentes ao $else$ é

[1]Uma linha de execução no espaço de estados é um possível caminho no espaço de estados do modelo que representa um comportamento específico do sistema. Não deve ser confundido com linha de execução de programas concorrentes.

[2]Caso nenhum *else* seja declarado, um *else* é automaticamente introduzido durante a tradução para o modelo final descrito na linguagem formal.

```
@behavior(
  ...
  choice{
    media > 7 : { ... }
    media == 7 : { ... }
    else : { ... }
  }
)
public String pegueSituacaoDoEstudante(Estudante estudante){
  float media = Estudante.getMedia();
  if (media>7)
    return "Acima da media.";
  else if (media == 7)
    return "Dentro da media.";
  else return "Abaixo da media.";
}
```

Código Anotado 3.9: Exemplo de anotação usando $choice$.

executado apenas se nenhuma das condições for verdadeira. Como as condições foram substituídas por $true$, em razão da abstração feita, o bloco de $else$ não seria executado jamais. Como a idéia é não deixar de refletir fluxo de execução algum no espaço de estados do modelo, o $else$ também deve ser substituído por $true$, possibilitando assim a execução de seu bloco correspondente.

```
@behavior(
  ...
  choice{
    true : { ... }
    true : { ... }
    true : { ... }
  }
)
public String pegueSituacaoDoEstudante(Estudante estudante){
        ...
}
```

Código Anotado 3.10: Exemplo de anotação usando $choice$ abstraindo variável.

A decisão sobre o que abstrair fica a cargo do programador, como acontece com qualquer linguagem de modelagem (modelador neste caso). Caso as propriedades que ele deseje verificar não dependam de forma alguma da existência de uma determinada variável, esta última pode ser eliminada do modelo comportamental.

Alguns comandos definidos em JaCA possuem exatamente a mesma semântica que a

linguagem de programação Java [3]. Estes comandos são seguidos de ponto e vírgula (;) e são explicados logo abaixo:

- $wait$ - usado para parar a linha de execução corrente temporariamente. A linha de execução é colocada num conjunto de linhas de execuções que foram paradas.

- $notify$ - usado para notificar a uma linha de execução que ela pode continuar sua respectiva tarefa de onde parou. Uma linha de execução do conjunto de linhas de execução, que foram adicionadas pela execução do comando $wait$, é escolhida aleatoriamente.

- $notifyall$ - usado para notificar todas as linhas de execução que elas podem continuar suas respectivas tarefas de onde pararam.

A anotação para modelagem de tratamento de exceções possui exatamente a mesma sintaxe de Java. Um exemplo de modelagem de tratamento de exceção é mostrado no Código Anotado 3.11. Esta parte da linguagem JaCA foi herdada de Java única e exclusivamente para suportar tratamentos de exceções. As abstrações feitas dentro das chaves das cláusulas try e $catch$ é que vão determinar o quanto o espaço de estados será grande.

Outra forma de modelagem existente é a anotação usando a estrutura $sync$*(<objeto>){ ... }*. A semântica desta estrutura é a mesma do bloco $synchronized$ da linguagem Java. A linha de execução, que estiver executando o bloco de instruções contido entre as chaves, manterá o *lock* do objeto representado por *<objeto>*. No Código Anotado 3.12, é apresentada esta anotação.

Com a linguagem JaCA é possível definir asserções sobre a execução do modelo do programa. Isto é feito através da estrutura $assert$*<condição>*. Caso uma asserção seja violada, o programador obtém uma seqüência de execuções do modelo que levam até tal violação. No Código Anotado 3.13, é mostrado o uso de uma asserção. Neste exemplo, caso o atributo $media$ do objeto $estudante$ seja negativo ou maior que 10 no momento em que a asserção for avaliada, ocorrerá uma violação no modelo.

[3] Em Java, o comando $wait$ pode possuir como parâmetro um período no qual a linha de execução esperará para ser notificada. Caso não seja notificada dentro deste período, a linha de execução retoma a execução de sua tarefa. Em JaCA, não há noção explícita de tempo. Desta forma, o comando $wait$ não recebe parâmetro.

```
@behavior(
  ...
  try{
    ...
  } catch(NullPointerException npe) {
    ...
  }
  ...
)
public float somarNotas(Estudante estudante){
  float result = 0;
  try{
    for (int i = 0; i < estudante.notas.length; i++) {
      result = result + estudante.notas[i];
    }
  } catch(NullPointerException npe) {
    this.enviarErro("Notas inexistentes!");
  }
  return result;
}
```

Código Anotado 3.11: Exemplo de anotação para tratamento de exceções.

A cláusula *assert* de JaCA pode ser utilizada na aplicação de *design by contract* no modelo comportamental do programa, analisando o estado de variáveis e de objetos antes e depois da invocação de métodos. Em outras palavras, pré- e pós-condições podem ser inseridas no modelo comportamental através da cláusula $assert$. Contudo, dependendo do que se quer garantir usando $assert$, esta cláusula pode não ser útil caso as asserções sejam feitas sobre variáveis compartilhadas por mais de uma linha de execução. Pois a intercalação entre tais linhas pode fazer com que um estado intermediário e indesejado de uma variável seja avaliado por uma cláusula *assert*.

Uma cláusula genérica definida na linguagem de anotação JaCA é a cláusula *task*. Esta cláusula pode ser usada para abstrair completamente uma porção de código. Esta abstração poderia ser feita de forma total simplesmente não sendo modelada. Entretanto, a vantagem de se usar a cláusula *task* é que o programador pode expressar sua execução nas propriedades a serem especificadas e verificadas. Apesar da abstração total ser feita, é possível verificar se há ou não execução num determinado fluxo de execução que contenha *task*. No Código Anotado 3.14, é mostrado um exemplo simples do uso desta cláusula. É importante saber que uma seqüência de $task;$'s equivale a um único $task;$, não importando a quantidade destes.

Devido à possibilidade de abstração de variáveis, a invocação de um método no modelo

```
@behavior(
  ...
  sync(estudante){
    ...
  }
  ...
)
public float somarNotas(Estudante estudante){
  ...
  synchronized(estudante){
    ...
  }
  ...
}
```

Código Anotado 3.12: Exemplo de anotação usando $sync$.

```
@behavior(
  ...
  Estudante estudante = new Estudante();
  ...
  assert ((estudante.media >=0) || (estudante.media <=10));
)
public static void main(String[] args) {
  ...
  Estudante estudante = new Estudante();
  ...
}
```

Código Anotado 3.13: Exemplo de anotação usando $assert$.

comportamental pode ser comprometida. Pois, uma variável usada como parâmetro por um método pode não existir no modelo. Para eliminar tal problema, além de a linguagem JaCA poder ser usada para abstrair o comportamento dos métodos, ela pode ser usada também para redefinir a assinatura dos métodos, bem como o retorno dos mesmos. Com isso, é possível propagar para dentro do escopo de um método a abstração feita fora de seu contexto.

Para indicar uma mudança no tipo de retorno ou na assinatura de um determinado método, a anotação deve ser iniciada com a cláusula $abstract$. No Código Anotado 3.15, devido a abstrações feitas no modelo, o tipo de retorno do método e sua assinatura são alterados. Caso o método que sofrerá alteração na assinatura ou no tipo de retorno seja sincronizado, deve-se mencionar isto em sua redefinição usando a palavra-chave $synch$. Além disso, se o método em questão for estático, a palavra chave $static$ tabém deve ser introdu-

```
@behavior(
  ...
  try{
    ...
  } catch(NullPointerException npe) {
    task;
  }
  ...
)
public float somarNotas(Estudante estudante){
  ...
  try{
    ...
  } catch(NullPointerException npe) {
    this.enviarErro("Notas inexistentes!");
  }
  ...
}
```

Código Anotado 3.14: Exemplo de anotação usando $task$.

zida na anotação, entre a cláusula que define sua visibilidade ($public$ ou $private$) [4] e o tipo de retorno.

```
@behavior(
  abstract public void determineGrau(){
    ...
  }
)
public String determineGrau(int anoCorrente){
  ...
}
```

Código Anotado 3.15: Exemplo de redefinição de método usando $abstract$.

Vale a pena salientar que não é relevante saber se determinada classe estende outra classe, ou se determinado método é estático. Estes tipos de informação são extraídos automaticamente do código pela ferramenta apresentada no Capítulo 4, sem intervenção alguma do usuário.

Além destas anotações ao nível de método, anotação de início e anotação comportamental, há também as anotações de especificação de propriedades. Estes tipos de anotação são definidos pela *tag @property*. Ela serve para determinar quais as propriedades que se de-

[4] Assim como os atributos são tratados, no modelo comportamental, não há distinção entre o que é publico ou privado. Mesmo assim estas cláusulas foram mantidas.

seja verificar no modelo anotado. A linguagem utilizada para especificar as propriedades é a linguagem LTL [Pnu77], que pode ser vista em detalhes no Apêndice A.

Este tipo de anotação é dividido em duas partes: declaração de proposiçoes e declaração de especificações. A declaração de proposições é definida pela sentença *propdec{ ... }*. As declarações de proposições ficam delimitadas pelas chaves e possuem o seguinte formato: *IdentificadorDaProposição :: ExpressãoBooleana*. Cada declaração de proposição deve ser finalizada com um ponto e vírgula (;).

As especificações são definidas pela sentença *spec{ ... }*. As propriedades em LTL também ficam delimitadas entre as chaves. Só podem ser usadas proposições que foram declaradas na sentença *propdec* dentro de uma mesma *tag @property*.

No Código Anotado 3.16, é mostrada a especificação de duas propriedades. A primeira propriedade especificada significa que *"sempre que a média do estudante for maior ou igual a 7, futuramente ele estará aprovado"*. A outra propriedade significa que *"futuramente o estudante estará aprovado ou reprovado"*. A primeira vista esta última propriedade pode parecer inútil. Contudo, por algum motivo, o atributo $aprovado$ do objeto $estudante$ pode não receber atribuição de valor booleano algum durante toda a execução do modelo. Assim, esta propriedade não seria satisfeita, pois a variável permaneceria nula durante todo o ciclo de execução do modelo.

```
@property(
  propdec{
    mediaSuficiente :: execestudante.media>=7;
    estudanteAprovado :: estudante.aprovado;
  }
  spec{
    [](mediaSuficiente -> <>estudanteAprovado);
    <>(estudanteAprovado || !estudanteAprovado);
  }
)
public static void main(String[] args) {
  ...
}
```

Código Anotado 3.16: Exemplo de especificação de propriedades usando *@property*.

As propriedades especificadas usando a *tag @property* são sensíveis ao contexto do método onde são anotadas, ou seja, elas não valem para todas as variáveis manipuladas durante a execução do modelo. Uma variável pode existir simplesmente durante a execução

de um método. Ao término da execução de tal método, a variável, declarada dentro do escopo do método, deixará de existir. Assim sendo, as propriedades especificadas no Código Anotado 3.16 dizem respeito ao objeto $estudante$ criado dentro do método $main$. Qualquer outro objeto criado no modelo, em qualquer outro método, não é levado em consideração na verificação desta propriedade, mesmo que o objeto possua identificador igual.

Como mencionado anteriormente, a cláusula $task$ funciona como uma abstração de qualquer porção de código. Caso uma parte do sistema esteja sendo representada no modelo pela cláusula $task$ e seja desejado especificar uma propriedade que possa envolver tal parte do modelo, o programador pode especificar uma propriedade usando a palavra-chave $taskExecutes$.

A propriedade ilustrada no Código Anotado 3.17 significa que em algum momento no futuro da execução do método $SomarNotas$, haverá alguma computação que é abstraída pela cláusula $task$, no modelo apresentado no Código Anotado 3.14, e que pode ser detectada e expressada pelas propriedades lógico-temporais.

```
@property(
  spec{
    <>(taskExecutes);
  }
)
public public float SomarNotas(Estudante estudante) {
  ...
}
```

Código Anotado 3.17: Exemplo de especificação de propriedades envolvendo $task$.

Vale a pena mencionar que as propriedades especificadas para um determinado método só valem para o nível de execução do mesmo. Tais propriedades não especificam o comportamento de métodos invocados dentro de outros métodos. Em outras palavras, as propriedades mostradas nos Códigos Anotados 3.16 e 3.17 só podem dizer respeito aos comportamentos dos métodos $main$ e $SomarNotas$, respectivamente. Caso algum método do objeto $estudante$ seja invocado, dentro de qualquer um dos métodos mencionados, suas propriedades devem ser escritas, caso seja desejado, em seus escopos.

Entretanto, as propriedades podem expressar mudanças de estados de objetos que são manipulados dentro do método onde são anotadas. Um exemplo disto foi mostrado no Código Anotado 3.16.

A ordem de chamadas e execuções de métodos também pode ser verificada. Há dois tipos de verificação neste sentido: *verificação de ordem de invocação de métodos* e *verificação de ordem de execução de métodos*. A primeira aborda os casos em que se deseja saber se um determinado método *mA()* é invocado antes de um certo método *mB()*, sem que o método *mA()* tenha sido necessariamente executado por completo. Para verificar se um certo método *mB()* é invocado apenas após a total execução de um outro método, é usado o segundo tipo.

No Código Anotado 3.18, é ilustrado um exemplo de verificação de ordem de execução e invocação de métodos. Neste exemplo, há declaração de duas proposições: *metodoGetMediaExecutado* e *metodoSetStatusInvocado*. A especificação descrita em LTL diz que *"o método setStatus da classe Estudante é invocado e o método getMedia, também da classe Estudante, é terminado somente após o próprio método getMedia ter sido totalmente executado"*. É claro que um método sempre será finalizado após ter sido totalmente executado. Esta propriedade serviu apenas para demonstrar o uso das cláusulas *exec*, *invoc* e *fim*.

```
@property(
  propdec{
    metodoGetMediaExecutado :: exec(Estudante.getMedia());
    metodoSetStatusInvocado :: invoc(Estudante.setStatus(bool));
    metodoGetMediaTerminado :: fim(Estudante.getMedia());
  }
  spec{
    [](metodoGetMediaExecutado ->
        (<>metodoSetStatusInvocado) && metodoGetMediaTerminado
      );
  }
)
public static void main(String[] args) {
  ...
}
```

Código Anotado 3.18: Exemplo de especificação de ordenação de chamadas de métodos.

O verificador de modelos Bogor gera o espaço de estados referente à execução do modelo comportamental do sistema. Contudo, o Bogor não é capaz de verificar a ordem de chamadas de métodos. Para que isso seja possível, são adicionadas variáveis booleanas globais referentes aos métodos envolvidos na seqüência a ser verificada. O funcionamento deste artifício e suas limitações são mostrados no Apêndice C.

3.3 Lidando com herança e *interface*

Diferentemente de JML, que lida com especificação comportamental de interfaces, JaCA não foi concebida para anotar interfaces. A linguagem de anotação aqui apresentada, foi feita para anotar métodos que possuam implementação. Estejam estes métodos contidos em classes concretas, abstratas ou estáticas (classes internas) [Joh06].

Em se tratando de herança, caso uma classe A possua métodos anotados, qualquer outra classe X que estenda esta classe, herdará também as anotações dos métodos herdados. O mesmo não ocorre quando a subclasse chega a sobrescrever algum método. Ou seja, quando um determinado método existente na superclasse é sobrescrito em uma subclasse, as anotações deste método devem ser feitas novamente. Isto é feito pois entende-se que o fato de haver *override* de um certo método, implica que a implementação de tal método numa determinada subclasse seja geralmente diferente da implementação do método contido na superclasse.

3.4 Usando *API* de terceiros

Geralmente, durante o desenvolvimento de um sistema, são utilizadas *api*'s desenvolvidas por terceiros. Quase sempre os desenvolvedores que usam estas *api*'s não possuem o código fonte das classes contidas na mesma. Às vezes possuem acesso ao código fonte, mas não estão interessados em conhecer e analisar o comportamento delas.

Caso não seja relevante para o que se deseja verificar, invocações aos métodos das *api*'s de terceiros podem ser abstraídas do comportamento do sistema. Entretanto, caso o desenvolvedor queira verificar a ordenação das chamadas de métodos ou qualquer outra coisa que dependa da modelagem da invocação de um determinado método do qual não se tenha acesso, ou não seja desejado conhecer, o código fonte, o desenvolvedor deve criar uma classe para "substituir" a da *api*.

Supondo que num determinado trecho de código de um método de uma classe implementada pelo próprio desenvolvedor, seja invocado um método chamado *metSemCodigo(int)*. Este método pertence a uma classe chamada *ClasseDeTerceiros*. Como não se tem acesso ao código fonte, ou não se deseja conhecê-lo, basta criar uma classe com o mesmo nome

contendo o método em questão. Este método não possui implementação algorítmica alguma e é classificado como *stub*. No Código Anotado 3.19, é ilustrada a criação de uma classe e de um método *stubs* e a anotação deste método.

```
public class ClasseDeTerceiros{
  ...
  @behavior(
    return 5;
  )
  public int metSemCodigo(int paramentro){
    return 5;
  }
}
```

Código Anotado 3.19: Criação de classe e método *stubs*.

Se for desejado abstrair o retorno do método *stub* em questão ou sua assinatura, basta usar a cláusula *abstract* como mostrado anteriormente. Um exemplo da abstração do retorno deste *stub* pode ser visto no Código Anotado 3.20.

```
@behavior(
  abstract public void metSemCodigo(int paramentro){
    task;
  }
)
```

Código Anotado 3.20: Abstração de retorno de método *stub*.

3.5 Conclusões

Como pode ser notado, a linguagem de anotação é bastante semelhante a Java. Daí pode surgir a pergunta: “por que não modelar com código Java”? A questão é que Java possui muitos recursos. Assim, seria necessário enumerar todas as restrições necessárias para que apenas um subconjunto de Java fosse usado. O usuário teria que ter em mente quais recursos poderiam ser ou não usados.

Além disso, pequenas mudanças na semântica de algumas estruturas seriam necessárias para atingir alguns objetivos. Como no caso de permitir várias maneiras de se modelar um laço, seria necessário mudar a semântica do $while$, por exemplo. Com isto, foi decidido por definir uma linguagem que se assemelhasse a uma linguagem de programação e que

tivesse uma semântica de execução também similar a linguagem de programação, mas não exatamente igual.

Outras linguagens de programação não orientadas a objetos (e.g., C e Pascal) também podem fazer uso da técnica e da linguagem de descrição comportamental apresentadas neste trabalho. Fazem-se necessárias apenas, algumas pequenas modificações nas estruturas da linguagem JaCA e algumas pequenas alterações nos algoritmos de conversão.

JaCA possui recursos como modelagem de declarações de atributos, desvios condicionais, laços, criação e instanciação de objetos, chamadas de métodos, tratamento de exceções e de blocos sincronizados. A linguagem aqui apresentada possui também uma cláusula para abstrair assinaturas e tipos de retorno de métodos, e uma cláusula para definição de asserções. Além disso, há a cláusula *task*, usada para abstração total de porções de código. Em se tratando de especificação de propriedades, há dois contextos: um para definição de proposições e outro para declaração das propriedades. Esta separação de contextos torna a especificação de propriedades mais organizada. Há também cláusulas usadas para especificar ordenação de chamada e execução de métodos.

A linguagem Java foi usada como uma instanciação para a prova de conceitos da técnica aqui apresentada. A linguagem aqui apresentada é capaz de descrever programas concorrentes orientados a objetos. Entretanto, uma limitação existente é o não suporte à herança múltipla, presente em linguagens orientadas a objeto como C++.

Na Tabela 3.1, é apresentada novamente a relação com as linguagens apresentadas no Capítulo 2 juntamente com a linguagem apresentada neste capítulo, a linguagem de anotação JaCA.

Os exemplos de anotações apresentados neste capítulo não tiveram o intuito de abstrair o comportamento dos métodos mostrados. As anotações apenas tiveram o propósito de apresentar o que é possível modelar usando a linguagem JaCA.

Caracter. / Ling.	ANNA	JML	AAL	TROLL	Perf. Lang.	BSL	JaCA
Linguagem alvo	Ada	Java	Java	OO	C++ Java Ada	Java	Java
Assemelha-se à linguagem alvo	√	√	√			√	√
Descrição comportamental				√	√		√
Suporta concorrência		√		√		√	√
Linguagem de anotação	√	√	√			√	√

Tabela 3.1: Tabela comparativa entre as linguagens, incluindo JaCA.

Capítulo 4

O *Plug-in*

Neste capítulo é apresentada a implementação da proposta de solução apresentada no Capítulo 1. A implementação da ferramenta aqui apresentada não objetivou produzir uma ferramenta para o uso imediato. Ela foi concebida como prova de conceitos da técnica definida neste trabalho. A ferramenta desenvolvida foi implementada no formato de *plugin* para o ambiente de desenvolvimento Eclipse [Ecl06], objetivando aproximar os processos de verificação e implementação. Com isso, além do modelo comportamental estar anotado junto ao código fonte formando um único artefato, o uso de um *plugin* como ferramenta centraliza as operações realizadas sobre tal artefato numa única ferramenta.

Primeiramente, é apresentada a arquitetura da ferramenta mencionando cada um dos componentes contidos na mesma e o papel deles, bem como os relacionamentos entre tais componentes. Logo em seguida são mostradas algumas telas do ambiente de desenvolvimento Eclipse com a ferramenta em funcionamento, destacando suas funcionalidades. Por último, são feitas algumas conclusões sobre a ferramenta, expondo suas limitações.

4.1 Arquitetura

Na Figura 4.1 é ilustrada a arquitetura da ferramenta. Por se tratar de um *plugin*, sua interface gráfica é o ambiente de desenvolvimento no qual está integrado.

Ao acionar o botão para verificar um determinado projeto de sistema, ou parte deste sistema, o *plugin* coleta todas as classes Java compiladas referentes aos pacotes ou classes selecionadas. O próprio ambiente Eclipse invoca o compilador Java para que este compile

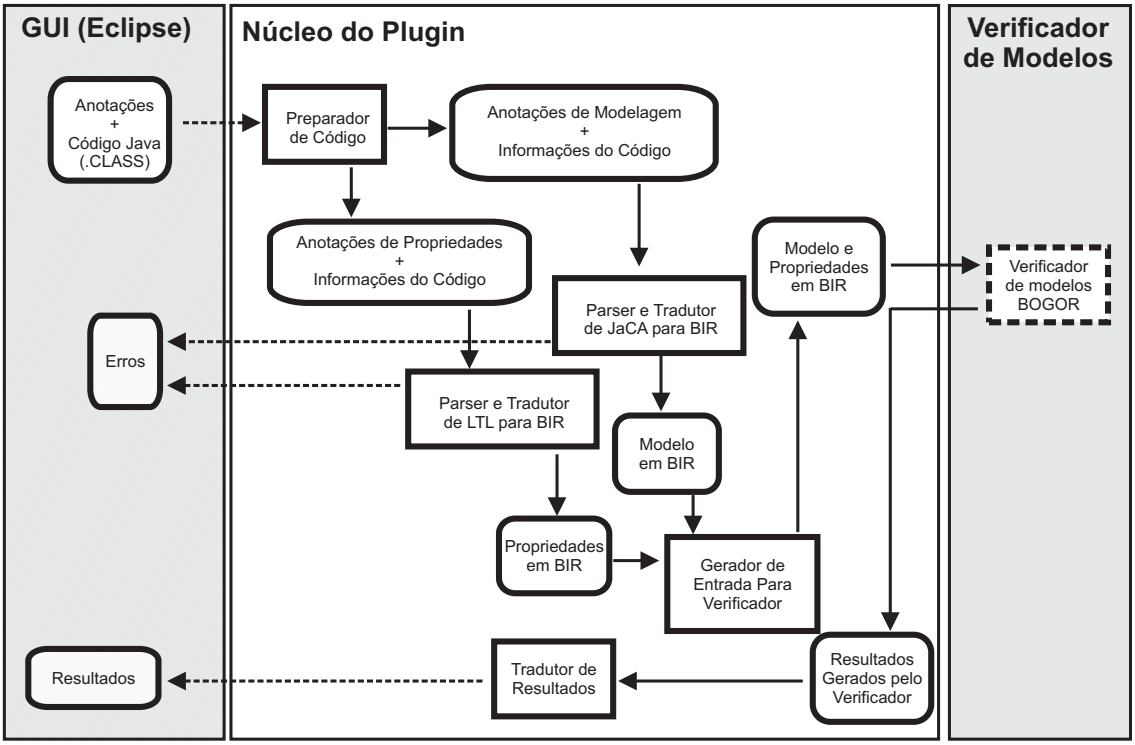

Figura 4.1: Arquitetura da ferramenta implementada.

todos os arquivos do projeto sempre que um dos arquivos Java for salvo pelo Eclipse. O *plugin* detecta onde os arquivos binários estão localizados e prossegue com o processo de verificação. A Figura 4.2 ilustra, em alto nível, como ocorre a interação entre o *plugin*, o Eclipse e o compilador Java. De posse das classes compiladas, um componente da ferramenta chamado *Preparador de Código* irá analisar todas elas, uma por vez. Através da propriedade de *reflexão*, são extraídas algumas informações sobre as classes e os métodos anotados. Estas informações serão necessárias para a construção do modelo comportamental que será posteriormente passado ao verificador de modelos.

Mais especificamente, o *Preparador de Código* produz dois artefatos, um com as *anotações de especificação de propriedades* e outro com as *anotações de descrição comportamental*. Cada uma das anotações presentes é coletada juntamente com informações relevantes sobre o código anotado (e.g., método sincronizado, tipo de retorno). As anotações de modelagem e as anotações de especificação de propriedades são passadas aos respectivos componentes responsáveis pelas suas análises sintática e semântica, *Parser e Tradutor de JaCA para BIR* e *Parser e Tradutor de LTL para BIR*. Após esta etapa, estes componentes traduzem as anotações de modelagem (anotações no formato da linguagem *JaCA*) e as de

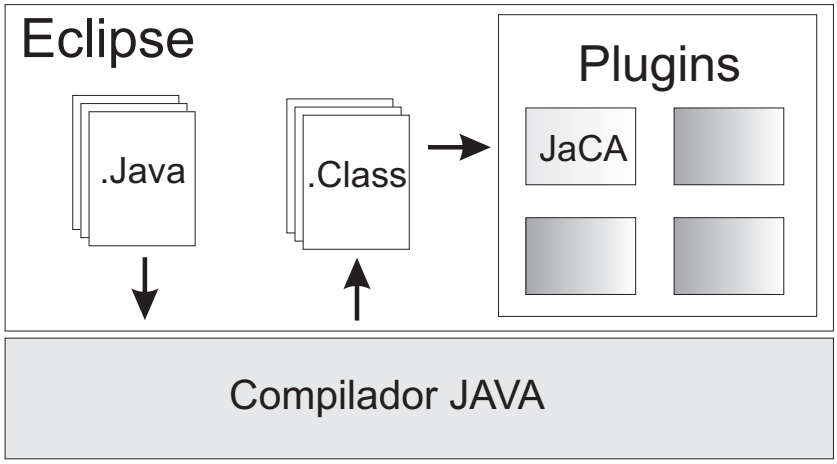

Figura 4.2: Interação entre o *plugin*, o Eclipse e o compilador Java.

especificação de propriedades (anotações no formato da linguagem LTL) para a linguagem de modelagem BIR e sua extensão para descrição de propriedades LTL, respectivamente.

Cada um desses componentes que analisam as anotações, *Parser e Tradutor de JaCA para BIR* e *Parser e Tradutor de LTL para BIR*, tem responsabilidades semelhantes. Em ambos é feita uma análise sintática para garantir a ausência de erros nas anotações. No caso de não haver erro algum nas anotações, estas são traduzidas para a linguagem de modelagem BIR, apresentada no Apêndice B. Caso haja algum erro em alguma das anotações, uma mensagem é exibida para o usuário por meio de uma *visão* do Eclipse chamada *Erros de Anotação*.

Entre as fases de análises sintática e semântica e a tradução para o código BIR, os componentes *Parser e Tradutor* transformam as anotações, juntamente com informações sobre o código fonte, em objetos Java. À medida que as análises sintática e semântica são feitas nas anotações de modelagem, objetos Java representando entidades BIR, como funções ou atributos, são criados. Na Figura 4.3 é mostrado o diagrama de classes referente às classes que representam tais entidades BIR. Para uma boa legibilidade, apenas os métodos e atributos mais relevantes para o processo de tradução são mostrados.

Como pode ser visto no diagrama de classes, todas as entidades possuem métodos do tipo *toBIR()*. São estes métodos que fazem a tradução da linguagem JaCA para a linguagem de modelagem BIR. Na verdade, após as análises sintática e semântica, as informações coletadas sobre as classes e os métodos são unidas com as anotações formando um artefato descrito num formato da linguagem JaCA com um aspecto mais próximo de BIR. Ou seja,

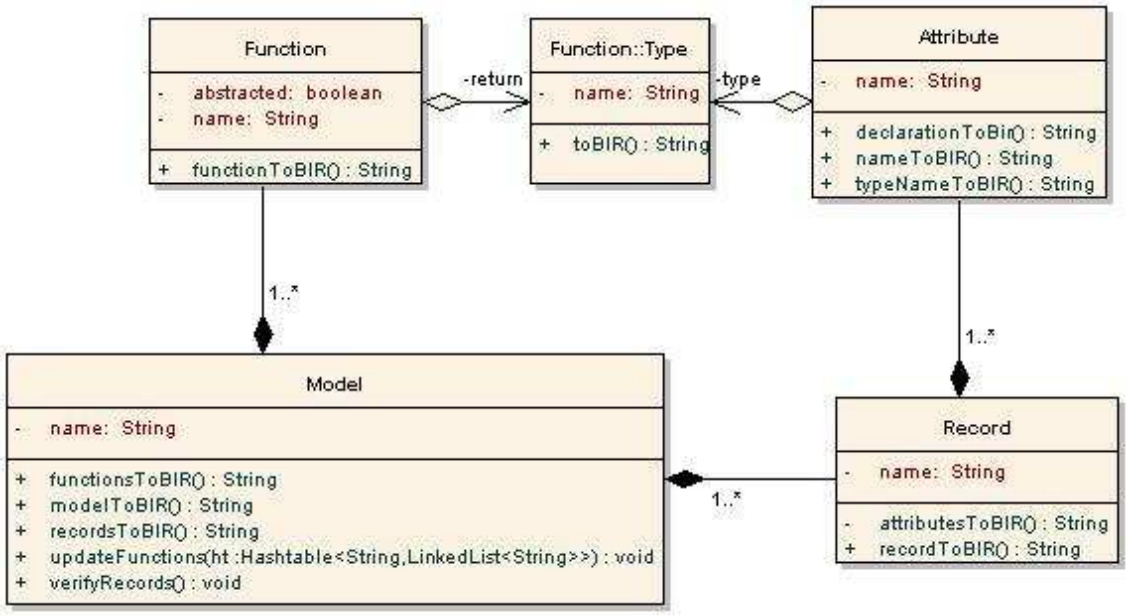

Figura 4.3: Classes referentes às entidades BIR.

um formato intermediário entre as linguagens JaCA e BIR é usado no processo de tradução. A partir deste formato é que os métodos *toBIR()*, geram o código final em BIR.

A classe $Function$, que representa as funções em BIR, possui atributos que são manipulados durante o processo de coleta e tradução das anotações e determinam se certo método Java deve ter sua assinatura e/ou retorno abstraídos de acordo com a anotação presente. A classe $Model$, que representa todo o modelo BIR, possui um método chamado *updateFunctions*. Tal método, através da propriedade de reflexão, coleta todos os tipos e os nomes dos parâmetros dos métodos representados por funções no modelo que não tiveram suas assinaturas redefinidas.

Outro método importante é o método chamado *verifyRecords()*, também da classe $Model$. Este método verifica se todas as classes, presentes nas anotações, usadas em declarações e instanciações de objetos e declarações de tipos de retorno de métodos, foram devidamente anotadas. Isto porque, a não anotação de alguma classe com a *tag @att* indica que a mesma não estará presente no modelo BIR.

Depois de terminado o processo de tradução das anotações de especificação de propriedades e das anotações de descrição comportamental para BIR, um componente simples, denominado *Gerador de Entrada Para Verificador*, junta os dois artefatos (no formato de BIR) num único artefato que será passado para o verificador de modelos.

Caso tenha ocorrido algum erro de anotação, o componente *Gerador de Entrada Para Verificador* não terá um dos artefatos BIR, anulando então o processo de verificação corrente. Caso tenha recebido os dois artefatos necessários, indicando que não houve erro algum nos processos de tradução, o modelo e as propriedades em BIR são passados ao verificador de modelos chamado Bogor, também apresentado no Apêndice B, que irá verificar se o modelo satisfaz ou não as propriedades definidas. Caso todas as propriedades sejam satisfeitas, uma mensagem positiva é retornada ao usuário. Em outro caso, se alguma das propriedades não for satisfeita, um ou mais contra-exemplos são gerados pela ferramenta Bogor e são coletados por um componente chamado *Tradutor de Resultados*.

Como os contra-exemplos são retornados no formato da linguagem BIR, tal componente tem como responsabilidade traduzir estes contra-exemplos para o formato da linguagem JaCA. Desta forma, fica mais compreensível para o desenvolvedor analisar os resultados, facilitando a localização do possível erro no código fonte, já que JaCA e Java são bastante semelhantes.

4.2 Telas e funcionalidades da ferramenta

Nesta seção são apresentadas algumas telas do ambiente Eclipse com o *plugin* instalado. Na Figura 4.4 é mostrada uma tela do Eclipse com a opção *Verificar Anotações* em foco. Ao ser instalado no ambiente Eclipse, o *plugin* dispõe de duas opções para verificar o sistema anotado: ir ao menu *JaCA* e em seguida selecionar a opção *Verificar Anotações*, ou clicar diretamente no ícone disponibilizado no painel de ícones do Eclipse.

Figura 4.4: Tela do ambiente Eclipse com o *plugin* instalado.

Tecnicamente, para usar o *plugin* para verificação de seus projetos, o programador precisa apenas conhecer a sintaxe da linguagem *JaCA*, apresentada no Capítulo 3. Com a sintaxe em

mente, é possível ao programador descrever o(s) modelo(s) do sistema em implementação. A ferramenta de verificação de modelos, acoplada à ferramenta aqui apresentada, age sobre o modelo detectando automaticamente *deadlocks* e verificando propriedades de segurança que podem ser expressas no formato de asserções. Ou seja, o programador precisa apenas usar a cláusula $assert$ apresentada no Capítulo 3 para definir algumas de suas propriedades de segurança.

Assim sendo, mesmo que o desenvolvedor não conheça a lógica temporal LTL, ele poderá verificar algumas propriedades de seu interesse fazendo uso da cláusula $assert$ para construir tais propriedades. Como por exemplo, o desenvolvedor poderia desejar verificar se após o final de uma determinada execução, um laço por exemplo, uma certa variável v ainda permanece nula: assert *v != null*. Caso a variável seja nula no momento em que o $assert$ é avaliado, uma falha é detectada no modelo e um contra-exemplo que mostre tal falha é retornado ao desenvolvedor no final da verificação.

Verificando o sistema ou partes dele

Para verificar um projeto anotado, basta que o programador selecione o projeto e clique em uma das opções mencionadas anteriormente: ir ao menu *JaCA* ou ir diretamente ao ícone referente ao *plugin*. Desta forma, todas as classes que contiverem anotações e estiverem dentro daquele projeto terão suas anotações convertidas num modelo comportamental do sistema, e tal modelo será verificado logo em seguida. Caso alguma propriedade lógico-temporal seja violada pelo modelo, um ou mais contra-exemplos, contendo uma seqüência de execuções cada, são retornados ao desenvolvedor. Cabe ao desenvolvedor analisar o contra-exemplo para detectar onde se encontra o erro nas anotações, e conseqüentemente no código fonte. Como as anotações que descrevem o comportamento do programa são bastante semelhantes ao próprio código fonte, o programador não terá a mesma dificuldade em detectar o problema em seu código de que teria se fosse uma outra linguagem formal usada na modelagem.

É possível que o desenvolvedor deseje verificar apenas uma certa parte do sistema. Para isto, basta que ele selecione um pacote em vez de selecionar o projeto inteiro. Na Figura 4.5 é mostrada a seleção de um pacote para ser verificado. Neste caso, todas as classes contidas no pacote *pacote1* e todas as classes contidas nos pacotes *pacote11* e *pacote12* terão suas anotações convertidas para o modelo correspondente àquela parte do sistema. Ao selecionar

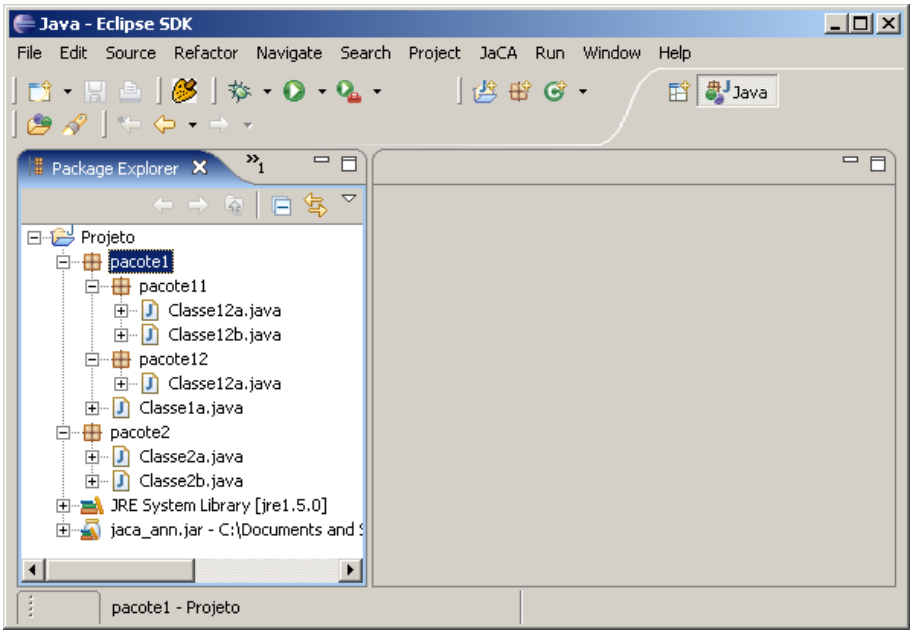

Figura 4.5: Selecionando um pacote para ser verificado.

um determinado pacote chamado *pacote2*, por exemplo, todas as classes que tiverem em seus pacotes o prefixo *pacote2* terão suas anotações convertidas. Selecionando o pacote *pacote11*, todas as classes que tiverem em seus pacotes o prefixo *pacote1.pacote11* terão suas anotações convertidas.

Com a possibilidade de se verificar partes do sistema, o desenvolvedor não é obrigado a anotar todo o código do sistema, ou o sistema não precisa estar completamente implementado. Contudo, selecionado um pacote ou mesmo uma classe, faz-se necessário que um método, e apenas um único método dentre todos os métodos anotados, possua a anotação da *tag @start*, indicando o ponto inicial de execução do modelo.

Esta abordagem de verificação em partes é bastante útil para uma verificação formal mais precisa. Já que apenas um subconjunto do sistema está sendo verificado, o espaço de estados gerado deverá ser menor do que no caso de se verificar o sistema por completo. Desta forma, é possível efetuar uma verificação mais precisa das partes do sistema de forma separada. É possível assim incluir no modelo algumas variáveis que, no caso de se verificar todo o sistema, deveriam ficar de fora para evitar a explosão do espaço de estados. Assim sendo, para uma verificação mais precisa do sistema sem que haja explosão do espaço de estados, o sistema deve ser dividido em partes a serem verificadas separadamente.

Erros no processo de anotação

Durante o processo de anotação (ou modelagem) do código fonte do sistema, o programador pode inserir algum erro não intencional nas anotações. Diferentes tipos de erro podem ser cometidos durante o processo de modelagem do sistema, tais como: invocação de método de um objeto não instanciado; invocação de método de um objeto cuja classe não tenha sido anotada[1]; uso de identificadores não válidos (e.g., variáveis não declaradas); parênteses, colchetes e chaves desbalanceados; entre muitos outros erros. Se o desenvolvedor cometer algum destes erros nas anotações de seu código, um aviso é passado a ele através de uma *visão* do Eclipse chamada *Erros de Anotação*.

Na Figura 4.6 é mostrado um exemplo de como o *plugin* trata o aviso destes erros de anotação. Neste exemplo, a intenção do desenvolvedor era a de anotar o código usando a

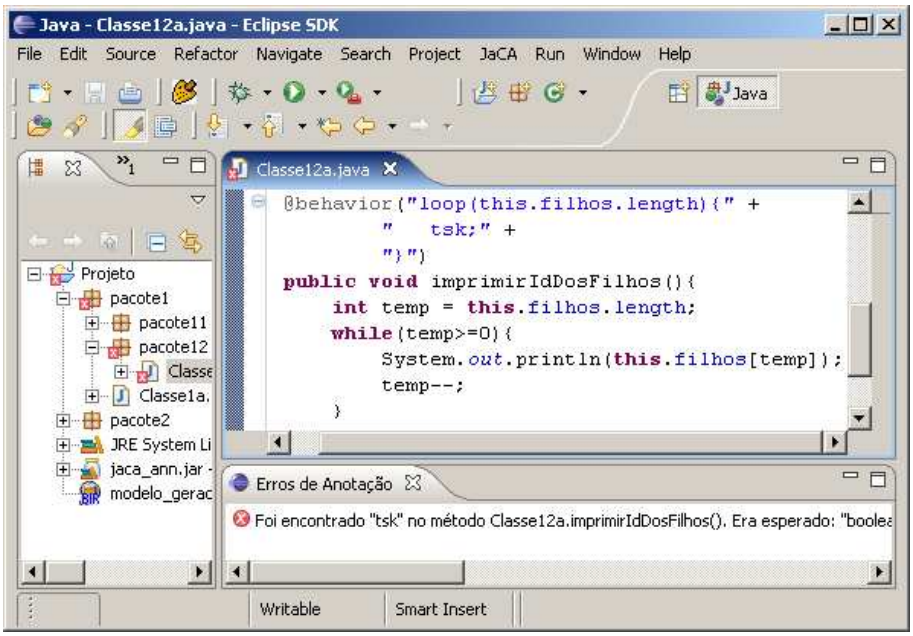

Figura 4.6: Erro presente dentro de anotação.

cláusula *task*, mas digitou *tsk* por engano. Ao acionar a opção de verificação, o *plugin* faz uma análise completa em todas as anotações a fim de garantir que elas estejam sintática e semanticamente corretas. Caso não haja nada de errado com as anotações a serem convertidas no modelo comportamental, a ferramenta continua o processo de verificação do sistema. Neste caso em particular, o *plugin* detecta que algo está errado nas anotações e mostra ao

[1]Caso uma classe (ou método) não seja anotada, ela não existirá no modelo final do sistema.

usuário as opções disponíveis de anotação para aquele caso, indicando em qual método e em qual classe se encontra o erro mostrado.

Um outro possível erro seria a inserção de uma anotação no lugar errado. Ou seja, anotar um método com a anotação da *tag @att*, que tem por natureza anotar classes. Na Figura 4.7 é mostrado um erro deste tipo. Uma tentativa de anotar uma classe com a anotação *@start*.

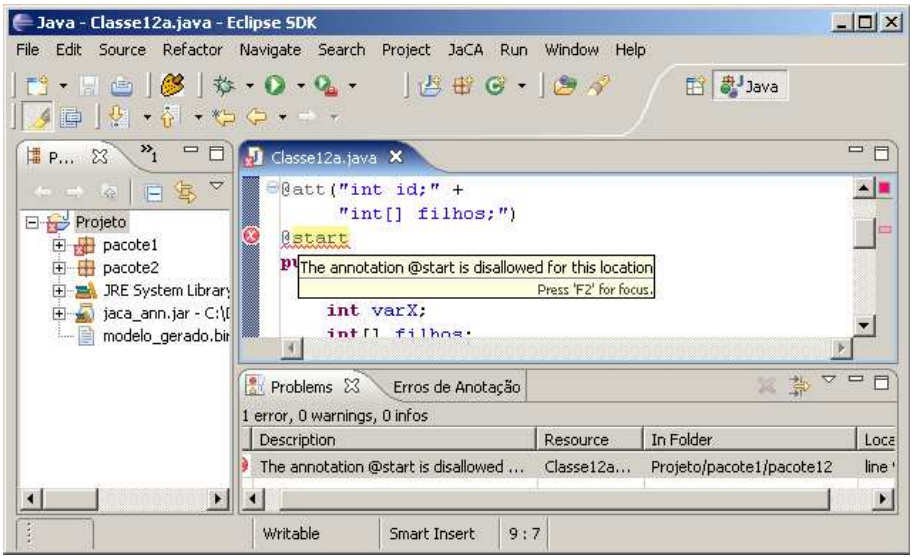

Figura 4.7: Anotação feita em lugar indevido.

Graças às especificações das anotações contidas no arquivo *JAR*, adicionado ao *Java Build Path* do projeto, o próprio Eclipse se encarrega de fazer este tipo de verificação.

Ao clicar na opção para verificar as anotações do sistema, o desenvolvedor deverá ter selecionado um projeto ou pelo menos algum pacote ou classe de algum projeto antes. Caso o desenvolvedor não faça uma seleção válida[2], ou seja, selecione qualquer coisa que não seja um projeto, um pacote ou uma classe Java, o *plugin* retorna uma mensagem de erro. Esta mensagem pode ser vista na Figura 4.8.

Resultados gerados com a verificação

Após a verificação do espaço de estados do modelo feita pela ferramenta de verificação acoplada ao *plugin*, um resultado é retornado ao usuário. Caso todas as propriedades especificadas, sejam as descritas em lógica temporal ou as descritas com a cláusula *assert*, sejam

[2]Uma seleção é válida se e somente se ela estiver em foco no momento em que o *plugin* for acionado.

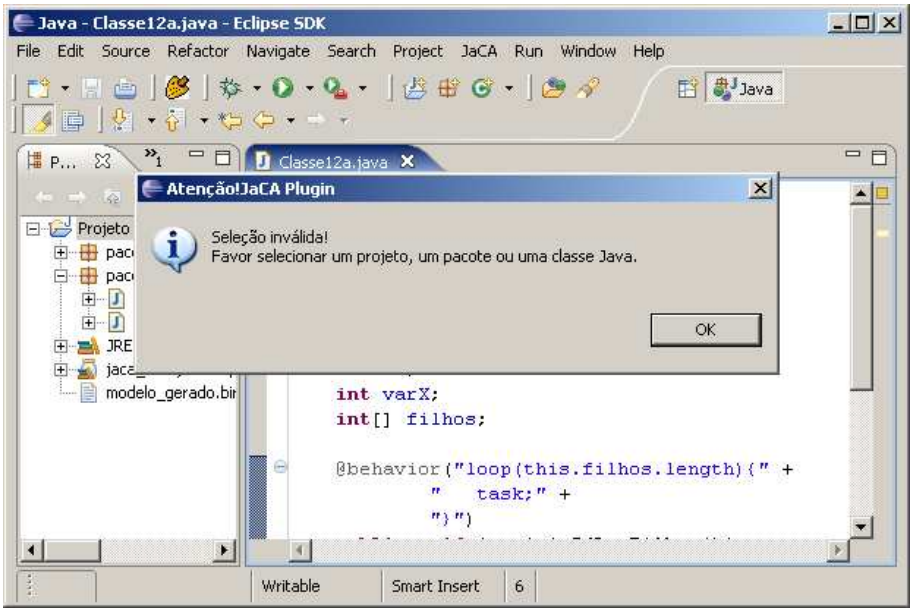

Figura 4.8: Erro de seleção para verificar o software.

satisfeitas pelo modelo, uma mensagem é retornada ao usuário indicando tal satisfação, como pode ser visto na Figura 4.9.

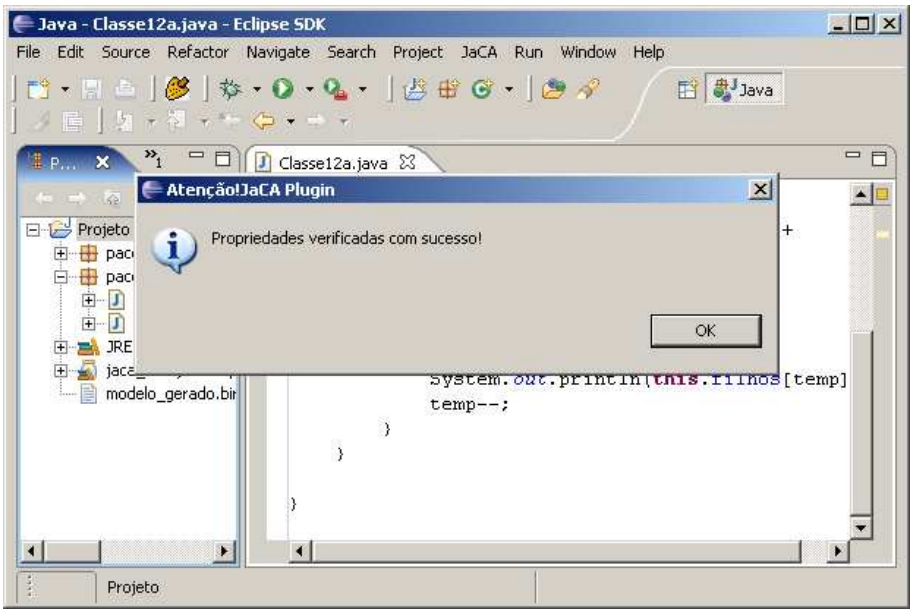

Figura 4.9: Mensagem de propriedades satisfeitas.

Caso contrário, se pelo menos umas das propriedades especificadas não satisfizer o modelo do sistema, uma mensagem indicando a(s) violação(ões) é exibida ao usuário, como pode ser visto na Figura 4.10. Além disso, um arquivo *TXT* é gerado com a(s) seqüência(s)

de execuções do modelo que levam ao(s) estado(s) que contradiz(em) a(s) propriedade(s) violada(s).

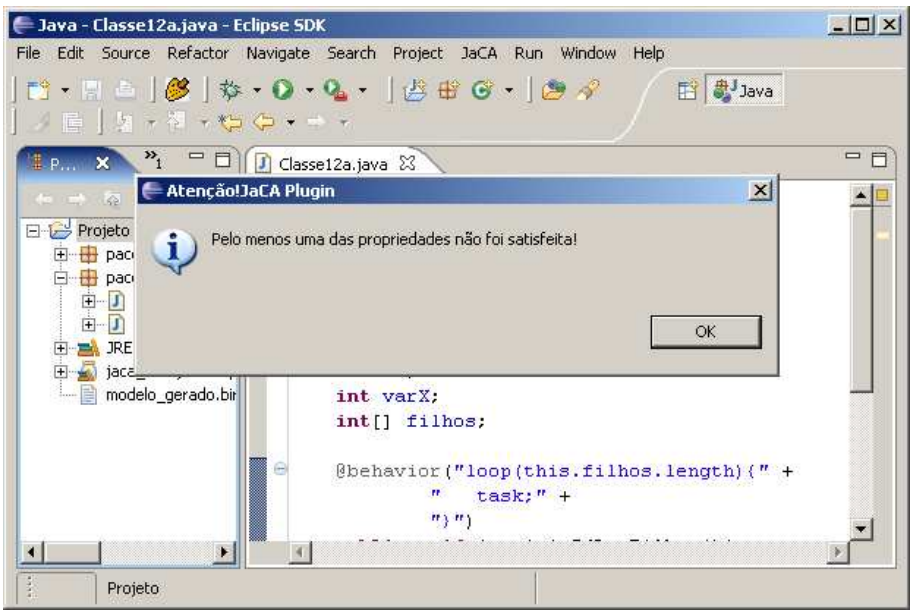

Figura 4.10: Mensagem de propriedades não satisfeitas.

4.3 Conclusões

Neste capítulo foi apresentada a implementação da solução definida neste trabalho. Como a ferramenta é integrada ao ambiente de desenvolvimento Eclipse, o usuário não precisa instalar uma aplicação a mais em sua estação de trabalho. Faz-se necessário apenas instalar o *plugin* no ambiente de desenvolvimento. Diferentemente de outras aplicações, o usuário não precisa editar e nem adicionar variáveis ao $CLASSPATH$. Basta apenas descompactar o *plugin* na pasta de *plugins* do ambiente Eclipse.

Há uma vantagem em se usar o Eclipse como plataforma para o *plugin* aqui apresentado. Cada vez que alguma alteração em qualquer parte de um projeto é salva, o Eclipse se encarrega de compilar todo o projeto. Desta forma, o usuário também não precisa se preocupar em compilar as classes Java sempre que for acionar o *plugin*, já que este recebe como entrada arquivos Java compilados.

O *plugin* mostrou-se bastante útil pois serve como uma *interface amigável* que aproxima o desenvolvedor leigo em métodos formais da técnica de *verificação de modelos*. A única

limitação neste uso está na especificação de propriedades que em parte é feita usando a linguagem LTL. Contudo, no Capítulo 6 é mencionado um possível trabalho futuro que deverá contornar tal limitação.

Uma limitação da ferramenta em si é ter como pré-requisito o compilador Java presente em sua versão 1.5. É esta versão de Java que possibilita a customização das anotações de código, que estão presentes como linguagens de modelagem e especificação. Contudo, qualquer código Java implementado em qualquer versão pode ser verificado, a única exigência é a presença do compilador 1.5, ou superior.

Capítulo 5

Experimentação e Validação

Neste capítulo são apresentados os experimentos realizados para validação do trabalho apresentado neste documento. Os experimentos foram realizados em dois sistemas concorrentes: um servidor de serviços para dispositivos móveis e um sistema para monitoração de energia. Em tais experimentos foram observados aspectos como: eficiência da técnica definida neste trabalho, funcionamento do *plugin* implementado para automatizar parte do processo, e facilidade no uso da linguagem de anotação também definida neste trabalho.

Como no sistema servidor de serviços, os processos de modelagem e especificação por meio das anotações foram feitos pelo autor do trabalho aqui apresentado, fez-se necessário um experimento que se fosse menos tendencioso ao apresentar os resultados. Desta forma, nos experimentos realizados com o sistema para monitoração de bateria, um programador fora do contexto do trabalho aqui apresentado fez uso da linguagem JaCA e do *plugin* implementado.

A seguir são apresentados os dois sistemas concorrentes estudados. Em cada um deles, é feita uma breve explanação sobre o sistema e seu comportamento, a partir de uma visão alto nível do mesmo. Logo em seguida, são mostrados os comportamentos que se espera que os sistemas possuam. Além disso, são mostrados os resultados obtidos pela verificação feita usando a técnica aqui definida. No primeiro sistema são mostradas também algumas anotações feitas no código fonte e alguns comentários sobre elas, tanto as anotações de modelagem quanto as de especificação de propriedades. Também são apresentados os resultados obtidos a partir da verificação do sistema usando as ferramentas apresentadas na Seção 2.2.3 para fins de comparação. E por fim, após a apresentação dos experimentos com os dois

sistemas, são apresentadas as conclusões gerais sobre este capítulo.

Todos os três aspectos, eficácia da técnica, funcionamento do *plugin*, e facilidade no uso da linguagem de anotação, foram estudados e analisados em todos os experimentos realizados, em ambos os sistemas estudados. Contudo, a validade da linguagem de anotação sofreu mais ênfase nos experimentos realizados no segundo sistema apresentado.

5.1 Servidor de Serviços

5.1.1 Apresentação

O sistema usado como primeiro estudo de caso se trata de um *servidor de serviços* para dispositivos móveis que usam uma conexão sem fio (*wireless*), mais especificamente *Bluetooth* [Blu01; BS00], para comunicação. O *servidor* fica constantemente esperando por requisições para conexões de clientes via a interface *Bluetooth*.

Bluetooth é uma tecnologia de rádio de curto alcance criada em meados da década de 1990. Esta tecnologia sem fio possibilita a transmissão de dados em curtas distâncias entre telefones, computadores e outros aparelhos eletrodomésticos. Mais do que somente uma substituição de cabos, a tecnologia sem fio *Bluetooth* provê uma conexão universal para redes de dados existentes, possibilitando a formação de pequenos grupos privados de aparelhos conectados entre si. A tecnologia de rádio do *Bluetooth* usa um sistema de freqüência de sinal que provê um *link* seguro e robusto, mesmo em ambientes com alto ruído e de grande interferência [Eri06].

Na Figura 5.1 é apresentado um esquema do cenário do *servidor de serviços* e de seus clientes. Todos os dispositivos móveis que suportam comunicação *Bluetooth* e estão dentro do raio de alcance do *servidor de serviços* podem fazer requisições por serviços.

Parte do comportamento deste servidor de serviços é ilustrada na Figura 5.2. Neste servidor podem ser registrados diferentes tipos de serviços. Ao ser iniciado, o servidor de serviços cria e inicia uma linha de execução para cada um dos serviços registrados em sua base de dados. *L.E.S1* corresponde à linha de execução do serviço 1, *L.E.S2* corresponde à linha de execução do serviço 2 e assim por diante, como ilustrado na figura. Cada uma destas linhas de execução permanece constantemente à espera por requisições de clientes para conexão ao

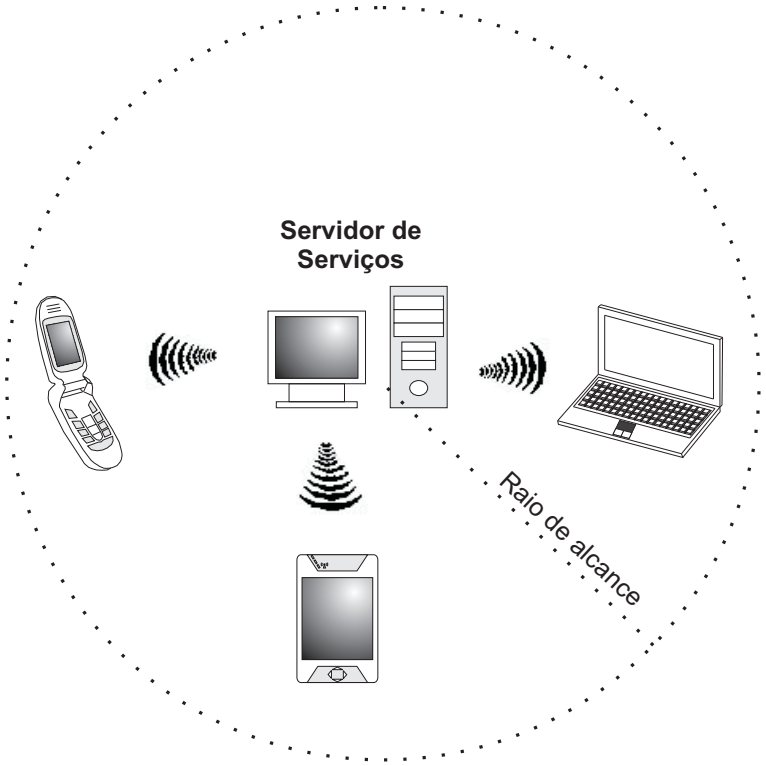

Figura 5.1: Comunicação entre aparelhos usando a tecnologia *wireless Bluetooth*.

serviço ao qual está associada.

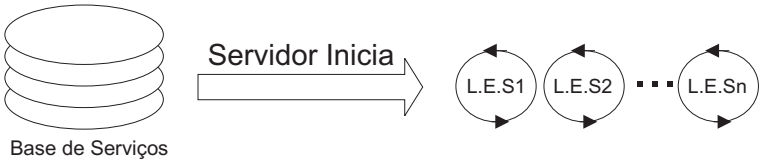

Figura 5.2: Criação de linhas de execução do servidor de serviços.

Sempre que uma das linhas de execução recebe uma requisição de um cliente para se conectar ao seu respectivo serviço, ela cria uma outra linha de execução que ficará atendendo às necessidades do cliente. Após criar esta linha de execução, a linha de execução principal do serviço volta ao seu estado de espera por novas requisições. Na Figura 5.3, duas requisições são feitas ao serviço *2*. A linha de execução do serviço *2*, denominada *L.E.S2*, cria uma nova linha de execução para cada uma das requisições, *L.E1.S2* para a requisição 1 e *L.E2.S2* para a requisição 2. Em seguida, volta ao seu estado anterior a chegada das requisições (espera por novas requisições). Cada uma destas duas linhas de execução atenderá somente ao seu respectivo cliente.

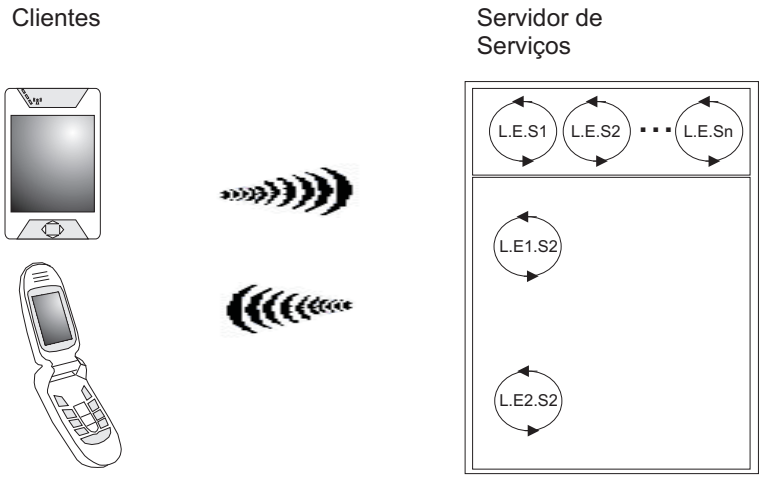

Figura 5.3: Criação de linhas de execução para atender clientes.

Comportamentos desejáveis

Abaixo são listadas partes do comportamento que se espera que o modelo comportamental do sistema possua:

1. Antes de sair do ar, o servidor deve finalizar todas as linhas de execução associadas aos serviços que por sua vez devem finalizar todas as linhas de execução que estão atendendo aos clientes. Todas as linhas de execução devem ser finalizadas "de baixo para cima". Ou seja, primeiro as linhas de execução que estiverem atendendo aos clientes, depois as que esperam por requisições e por último o servidor.

2. Ao mandar finalizar uma linha de execução que está atendendo a um cliente, esta linha de execução deve mandar uma mensagem para o cliente avisando que o servidor foi desligado. A linha de execução só pode ser efetivamente finalizada após esta mensagem ter sido enviada para o cliente.

3. Ao receber uma mensagem de "fim de serviço" do cliente, a linha de execução que o está atendendo deve ser finalizada.

4. Em algum momento, uma linha de execução criada para atender determinado cliente deve terminar sua tarefa e deve ser finalizada.

Tais comportamentos foram verificados por meio de propriedades especificadas sobre valores de atributos e chamadas a métodos pertencentes a três das oito classes que compõem

o sistema. Na Figura 5.4 é apresentado um diagrama de classes simplificado do sistema, métodos e atributos são omitidos. As três classes envolvidas nas propriedades especificadas para garantir os comportamentos mencionados são: *ServerThread*, *ServiceDispatcher* e *BluetoothConnection*. A primeira classe representa a linha de execução do serviço que espera por requisições. A segunda classe representa a linha de execução criada para atender a um determinado cliente. E a terceira e última classe representa o canal de comunicação entre a linha que atende ao cliente e o cliente atendido.

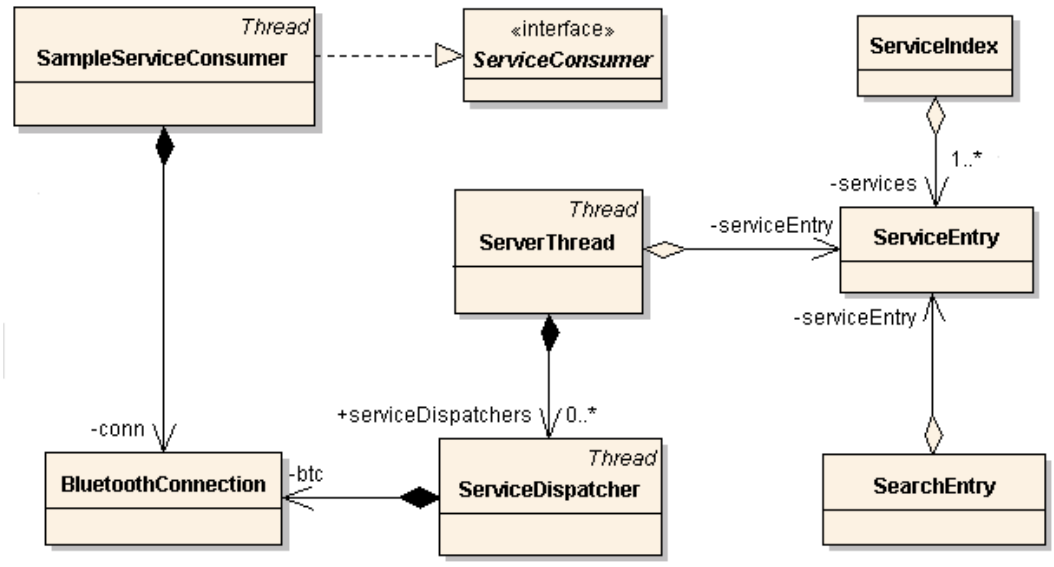

Figura 5.4: Diagrama de classes do servidor de serviços.

5.1.2 Anotações do Código

Nesta seção são mostradas algumas anotações de modelagem feitas no código fonte do servidor de serviços. Também são mostradas todas as anotações de especificação de propriedades feitas. Alguns comentários sobre as anotações são apresentados. No Apêndice E, é mostrado todo o restante das anotações e também todo o código do sistema.

Anotações comportamentais

Na técnica de verificação de modelos, o modelo do sistema é que é verificado, e não o sistema. Este modelo nada mais é do que uma abstração comportamental deste sistema. As anotações comportamentais funcionam como o modelo do código anotado. Desta forma,

elas não expressam, e nem devem expressar, exatamente todo o comportamento do código, mas sim uma abstração dele, tornando viável o processo de verificação.

No Código Anotado 5.1 é mostrado o código do método *main* juntamente com sua anotação. Devem ser observadas algumas coisas. A primeira e mais importante é que as anotações, neste código anotado, diferem do código fonte. Esta diferença entre código e modelo não é feita ao acaso. São declaradas variáveis (*st1*, *st2* e *st3*) para que seja possível referenciar os serviços nas especificações em LTL. E, em vez de iniciar o modelo do servidor com inúmeros serviços, nesta anotação o modelo é iniciado apenas com três. Isto devido ao conhecido problema da explosão do espaço de estados. Mais detalhes sobre tal problema nesta verificação são mencionados mais à frente no formato de dados obtidos a partir das verificações.

```
1  @property( ... )
2  @behavior(
3    ServiceEntry[] services = ServiceIndex.allServices();
4    ServerThread st1 = new ServerThread(services[0]);
5    ServerThread st2 = new ServerThread(services[1]);
6    ServerThread st3 = new ServerThread(services[2]);
7    start{ st1; st2; st3; }
8    invokeService(st1);)
9  @start
10 public static void main( String[] args ) {
11   ServiceEntry[] services = ServiceIndex.allServices();
12   ServerThread thread = new ServerThread( services[0] );
13   for( int i = 1 ; i < services.length ; i++ ) {
14     thread = new ServerThread( services[i] );
15     thread.start();
16   }
17   invokeService(thread);
18 }
```

Código Anotado 5.1: Método principal que inicia o sistema.

De acordo com a anotação presente no Código Anotado 5.1, são instanciadas três *threads* de serviços (linhas 4, 5 e 6), cada uma correspondente a um serviço registrado na base de serviços, que serão iniciadas (linha 7) juntamente com o início de execução do servidor. Originalmente, o modelo e o sistema só cobriam o comportamento e a execução do servidor sem que houvesse requisições de serviços. Foi então criado um método para simular a invocação de determinado serviço, a fim de verificar propriedades em cima do comportamento estimulado por esta requisição (linha 8).

Há também, neste código anotado, duas outras anotações: *@start* e *@property*. A primeira (linha 9) indica que o método em questão deve ser o ponto inicial de execução do modelo comportamental a ser gerado. A segunda (linha 1) contém as especificações de propriedades que serão mostradas mais adiante.

No Código Anotado 5.2 é apresentada a anotação do método que finaliza o servidor, classe *ServerThread*. A finalização do servidor é indicada com a atribuição do valor *false* à variável *serverIsUp* (linha 10).

```
1  @behavior(
2    int i=-1;
3    loop(serviceDispatchers.size()){
4      choice{
5        (serviceDispatchers[++i] != null): {
6          serviceDispatchers[i].abortService();
7        }
8      }
9    }
10   this.serverIsUp = false;
11   service.close();)
12 public void abortServer() {
13   for( int i = 0 ; i < serviceDispatchers.size() ; i++ ) {
14     ServiceDispatcher dispatcher =
15       (ServiceDispatcher)serviceDispatchers.elementAt( i );
16     if( dispatcher != null ) { dispatcher.abortService(); }
17   }
18   try {
19     serverIsUp = false;
20         service.close();
21   } catch( IOException e ) { e.printStackTrace(); }
22 }
```

Código Anotado 5.2: Método *abortServer* da classe *ServerThread*.

No servidor de serviços, há eventuais trocas de mensagem entre o cliente e a linha de execução que o atende. Muitas dessas mensagens são manipuladas como *string* pela classe *BluetoothConnection*. Como deve ser aplicado um nível de abstração para se construir um modelo que não cause a explosão do espaço de estados, alguns métodos desta classe foram totalmente abstraídos. No Código Anotado 5.3 são mostrados a anotação e o código de um método que faz computações sobre uma *string*. Como ainda é desejado verificar chamadas de métodos, este é anotado com a mínima informação possível, cláusula *task* (linha 3). A assinatura do método, bem como seu parâmetro, foi abstraída (linha 2).

```
1  @behavior(
2    abstract private void getFullMessage(){
3      task;
4    })
5  private String getFullMessage( Vector fullMessage ) {
6    String fullMessageAsString = "";
7    for( int i = 0 ; i < fullMessage.size() ; i++ ) {
8      String partialMessage = (String)fullMessage.elementAt( i );
9      fullMessageAsString += partialMessage;
10   }
11   return fullMessageAsString;
12 }
```

Código Anotado 5.3: Abstração total de manipulação de *strings*.

Anotações de especificação de propriedades

Baseando-se nas propriedades enumeradas anteriormente, são apresentadas nesta seção as propriedades especificadas e verificadas junto ao modelo gerado a partir das anotações feitas. Para tornar as especificações mais legíveis, cada uma das propriedades é mostrada separadamente.

Propriedade 1

Primeiramente, é necessário declarar as proposições que serão usadas nas especificações lógico-temporais. Estas proposições são ilustradas na Figura 5.5.

```
1  @property(
2    propdec{
3      InvocaFinalizarServidor :: invoc(ServerThread.abortServer());
4      ExecutaFinalizarServidor :: exec(ServerThread.abortServer());
5      ServDoClienteAbortado :: serviceDispatchers[0].serviceAborted == true;
6      ServDoServidorAbortado :: st1.service == null;
7      ServidorAbortado :: st1.serverIsUp == false;
8    }
9    ...
10 }
```

Figura 5.5: Declaração de proposições para propriedade 1.

As proposições das linhas 3 e 4 dizem respeito à invocação e execução total do método *abortServer()*. Desta forma, é possível raciocinar sobre propriedades que envolvem trocas de mensagens. Já as linhas 5, 6 e 7 dizem respeito à finalização das linhas de execução. A

primeira se refere à finalização da linha de execução que atende ao cliente. A segunda se refere à finalização da linha de execução que aguarda por requisições e cria novas linhas que atendem clientes. Já a terceira, refere-se a finalização do próprio servidor de serviços.

Duas especificações foram construídas a partir destas proposições. A primeira delas é:

```
[](InvocaFinalizarServidor ->
   <>(ServDoClienteAbortado ^ <>ServDoServidorAbortado)
);
```

Esta especificação diz que *"sempre que for invocado o método para finalizar o servidor, a linha de execução que atende ao cliente e a linha de execução do serviço que espera por requisições serão finalizadas nesta ordem"*.

A segunda especificação é a seguinte:

```
[](ExecutaFinalizarServidor ->
  (ServidorAbortado && ServDoServidorAbortado && ServDoClienteAbortado)
);
```

Ela diz que *"sempre que o método para finalizar o servidor for completamente executado, o servidor, a linha de execução que atende ao cliente e a linha de execução do serviço que espera por requisições já terão sido finalizados"*. Estas duas especificações garantem a primeira propriedade.

Propriedades 2, 3 e 4

Novamente, em primeiro lugar, é necessário declarar as proposições que serão usadas nas especificações lógico-temporais. Estas proposições são ilustradas na Figura 5.6.

As proposições declaradas nas linhas 3 e 4 dizem respeito a invocação e a execução total do método *abortService()* da classe *ServiceDispatcher*. Já nas linhas 5 e 6, são declaradas as proposições referentes à invocação e a execução total do método *transmitDiscardClientMessage()* da classe *BluetoothConnection*. E por fim, na linha 8, é declarada a proposição referente à finalização da execução do método *abortService()* da classe *ServiceDispatcher*.

Foram descritas quatro propriedades especificadas usando estas proposições. Para a propriedade 2, foram usadas as especificações:

```
[](InvocaAbortarSevico -> <>IniciaTransMsg);
[](!MsgTransmitida -> !ExecutaAbortarSevico);
```

```
1  @property(
2    propdec{
3      InvocaAbortarSevico :: invoc(ServiceDispatcher.abortService());
4      ExecutaAbortarSevico :: exec(ServiceDispatcher.abortService());
5      IniciaTransMsg :: invoc(BluetoothConnection.transmitDiscardClientMessage());
6      MsgTransmitida :: exec(BluetoothConnection.transmitDiscardClientMessage());
7      MsgFimDeServico :: exec(BluetoothConnection.messageWithoutTail());
8      FinalizaAbortarSevico :: fine(ServiceDispatcher.abortService());
9    }
10   ...
11 )
```

Figura 5.6: Declaração de proposições para propriedades 2, 3 e 4.

A primeira significa que *"sempre que for invocado o método para finalizar o serviço, futuramente será iniciada a transmissão da mensagem para o cliente com um aviso"*. Esta mensagem por si só não garante a segunda propriedade. Para garanti-la, tem-se a segunda especificação que significa que *"sempre que a mensagem não tiver sido completamente transmitida ao cliente, o método de finalização do serviço não terá sido completamente executado"*.

Para garantir a terceira e quarta propriedades, tem-se as especificações:

```
[](MsgFimDeServico -> <>FinalizaAbortarSevico);
<>(ExecutaAbortarSevico);
```

A primeira significa que *"sempre que for completamente enviada a mensagem de fim de serviço, o método para abortar serviço será finalizado"*. E a segunda, *"o método para abortar serviço será executado em algum momento no futuro"*.

Outras propriedades

Outras propriedades foram especificadas, baseadas nas seguintes proposições apresentadas na Figura 5.7. As proposições declaradas nas linhas 3, 4 e 5 dizem respeito ao estado das linhas de execução que esperam por requisições de clientes. Cada uma destas proposições indica que a linha está em execução. As demais proposições, linhas 6, 7 e 8, indicam que as mesmas linhas de execução não estão mais em funcionamento.

Baseando-se nestas proposições, as seguintes especificações são descritas:

```
[](serv1Ok -> <>serv1Finished);
```

```
@property(
  propdec{
    serv1Ok :: st1.serverIsUp;
    serv2Ok :: st2.serverIsUp;
    serv3Ok :: st3.serverIsUp;
    serv1Finished :: !st1.serverIsUp;
    serv2Finished :: !st2.serverIsUp;
    serv3Finished :: !st3.serverIsUp;
  }
  ...
}
```

Figura 5.7: Outras de proposições declaradas.

```
[](serv2Ok -> <>serv2Finished);
[](serv3Ok -> <>serv3Finished);
```

Cada uma delas significa que *"sempre que o serviço entrar no ar, futuramente ele será finalizado"*. Em outras palavras, um serviço jamais ficará no ar para sempre.

5.1.3 Resultados

Primeiramente foram definidos dez cenários. Estes cenários são divididos em três grupos: um grupo com apenas um serviço no ar (A), outro grupo com dois serviços no ar (B), e um terceiro grupo com três serviços no ar (C). Como as linhas de execução não compartilham recursos, poderiam ter sido verificados apenas os cenários do primeiro grupo. Contudo, foram adicionadas mais linhas de execução para que fosse analisado o impacto real das intercalações, provocadas por mais de uma linha de execução, no processo de verificação.

O grupo A contém três cenários: um sem requisição, um com uma requisição ao serviço no ar, e outro com duas requisições ao serviço no ar. O grupo B contém três cenários: um sem requisição de serviço, um com requisição a apenas um dos serviços no ar, e outro com requisição aos dois serviços no ar. E, analogamente ao grupo B, o grupo C contém quatro cenários: um sem requisição, um com uma requisição a um dos serviços no ar, outro com requisição a dois serviços distintos e outro com requisição a todos os serviços no ar.

Como só foi encontrado um erro de implementação no modelo, um outro erro foi introduzido propositalmente no código, e refletido no modelo, para verificar se a técnica seria capaz de detectá-lo. Além dos cenários mencionados, foram definidos outros três cenários,

um em cada grupo. Tal erro, sugerido pelo próprio desenvolvedor, faz com que a linha de execução de determinado serviço não mantenha referências às linhas de execução criadas por ela para atender aos clientes requisitantes. Desta forma, ao ser finalizado o servidor, o cliente não recebe aviso de finalização. O serviço simplesmente pára de funcionar. Nos três cenários, contendo o erro inserido, foi feita uma requisição a apenas um dos serviços no ar.

Nem todos os cenários definidos foram verificados. Alguns deles foram excluídos por que cenários considerados mais simples que eles não puderam ser completamente verificados. Eles foram excluídos da verificação devido à explosão do espaço de estados. Esta explosão não ocorreu exatamente por causa do número de estados, mas sim pelo tamanho de cada estado armazenado em memória. A seguir são mostrados os resultados para cada um dos cenários verificados totalmente e parcialmente e quais cenários não foram verificados.

Para todos os experimentos, foi usado um computador dedicado exclusivamente para executar o processo de verificação [1]. O computador possuía a seguinte configuração: processador *Athlon XP* com 1.6 GHz e com 1GB de memória RAM DDR (266MHz).

Primeiro grupo - Um serviço no ar

No primeiro cenário, foi feita uma verificação sem requisição de serviço. Nenhum problema de implementação foi encontrado. Apenas algumas propriedades que não foram especificadas corretamente reportaram erros.

As propriedades que reportaram “erros” foram aquelas especificadas usando as proposições mostradas na Figura 5.7. Estas propriedades significam que sempre que uma linha de execução, que espera por requisições, entrar no ar, ela será finalizada num momento futuro. Como não há em momento algum, no modelo ou no código, execução do trecho responsável pela finalização destas linhas de execução, as propriedades não são satisfeitas. Como não é intenção parar os serviços, neste caso específico, isto não representa um erro na implementação.

No segundo cenário foi feita uma requisição de serviço para o serviço que estava no ar. Já que não foi possível usar dispositivo móvel algum para requisitar o serviço ao servidor, foi implementado um método chamado *invokeService(ServerThread)*. Após requisição e

[1] No verificador de modelos Bogor, a verificação é feita a medida que o espaço de estados é construído, ou seja, *on-the-fly*.

execução do serviço, uma mensagem finalizando o uso do mesmo é enviada.

Nenhum problema foi encontrado. Até as propriedades que reportaram os falsos erros foram satisfeitas desta vez, pois o serviço em questão é finalizado.

O terceiro cenário, com duas requisições a um mesmo serviço, indicou um erro. A linha de execução principal do serviço fica bloqueada pelo método $StreamConnectionNotifier.acceptAndOpen()$ aguardando por requisições. Ao chegar uma requisição, ela é desbloqueada e inicia o processo de criação da linha de execução que ficará atendendo ao cliente que requisitou o serviço. Enquanto este processo não acaba, a linha de execução principal não fica aguardando por novas requisições. Desta forma, ao chegar uma segunda requisição durante o processo mencionado, a linha de execução principal não será capaz de lidar com ela, ocasionando um erro.

O quarto cenário definido foi aquele com o erro inserido propositalmente. Ao ser criada a linha de execução que atenderá ao cliente, uma referência a esta linha é guardada num *array* para que se possa enviar uma possível mensagem de finalização do servidor. O erro inserido no sistema foi, como mencionado anteriormente, a eliminação da guarda da referência. Na Figura 5.8 é ilustrada parte do código da linha de execução que espera por requisições de clientes. Na linha 1, a linha de execução que atenderá ao cliente é criada. A linha 2, que é excluída para inserir o erro, representa a guarda da referência mencionada. E por último, na linha 3, é iniciada a execução da linha que atenderá ao cliente. Como a linha 2 não é relevante ao funcionamento de nenhuma linha de execução, o defeito não é detectado facilmente. Baseado na inserção deste erro, as propriedades 1 e 2 não foram satisfeitas. Apesar da linha de execução que atende ao cliente ser finalizada por falta de comunicação com o servidor, este último não é capaz de finalizá-la corretamente. Isto faz com que o cliente tenha seu serviço finalizado de forma inesperada, sem receber a mensagem de finalização.

```
1 ServiceDispatcher dispatcher = new ServiceDispatcher();
2 serviceDispatchers.addElement( dispatcher );
3 dispatcher.serveRequest( btc );
```

Figura 5.8: Erro inserido deforma proposital.

A terceira propriedade foi satisfeita por que há uma restrição de *fairness* (mais detalhes no Apêndice A). Assim, apenas os caminhos em que a mensagem de fim de serviço foi enviada são avaliados. Neste caso, nenhum caminho. Já a quarta propriedade (*futuramente*

o método para abortar serviço será executado) não foi satisfeita. O modelo descrito nas anotações é mostrado no Código Anotado 5.4. O cliente pode querer usar o serviço para sempre, dada a inclusão da condição *true* no laço (linha 2).

```
1  @behavior(
2    loop(true | this.clientWantsService){
3      ...
4      this.clientWantsService = false;
5    })
6  public void run() {
7    ...
8    while( clientWantsService ) {
9      ...
10   }
11   ...
12 }
```

Código Anotado 5.4: Anotação de requisição de serviço.

Os dados referentes ao número de estados gerados, número de transições geradas e tempo gasto em cada um dos cenários deste grupo podem ser visto na Tabela 5.1. O espaço de estados referente ao cenário com duas requisições não foi completo por que, assim que o erro foi encontrado, o processo de verificação foi interrompido.

Cenário	Estados	Transições	Tempo	Espaço de Estados
Sem requisição	73	95	< 1s	Completo
Uma requisição	3.888	11.145	17s	Completo
Duas requisições ao mesmo serviço	20.747	57.896	1m22s	Parcial
Uma requisição com erro proposital	3.821	11.065	15s	Completo

Tabela 5.1: Dados obtidos no grupo A.

Segundo grupo - Dois serviços no ar

Similarmente ao primeiro grupo, foi feita uma verificação sem requisição alguma de serviço. Nenhum problema foi encontrado. Exceto, é claro, pelas propriedades que expressavam a eventual finalização das linhas de execução, Figura 5.7.

No segundo cenário, uma requisição foi feita a um dos serviços. Novamente todas as propriedades foram satisfeitas. O tempo necessário para a verificação deste cenário, em comparação ao tempo do segundo cenário do primeiro grupo, mostra como a presença de linhas de execução aumenta o espaço de estados. Não foi verificado cenário com duas requisições a um mesmo serviço por que cenário equivalente foi verificado no primeiro grupo.

No terceiro cenário foi feita uma requisição a cada um dos serviços postos no ar. O espaço de estados resultante não pôde ser completamente gerado. As propriedades especificadas foram satisfeitas. Contudo não se pode afirmar que elas (as propriedades) são satisfeitas por todo o modelo. Como apresentado na Tabela 5.2, o tempo necessário à verificação aumenta exponencialmente junto com a complexidade do cenário. O cenário com uma requisição a um serviço demandou aproximadamente 16 minutos, enquanto o cenário com uma requisição a cada um dos dois serviços demandou aproximadamente 12 horas.

No quarto cenário, com erro inserido, foi feita a requisição de apenas um serviço. Novamente as propriedades 1 e 2 não foram satisfeitas. A Tabela 5.2 mostra os dados obtidos nos experimentos realizados neste segundo grupo.

Cenário	Estados	Transições	Tempo	Espaço de Estados
Sem requisição	1.332	3.300	3s	Completo
Uma requisição a um dos serviços	176.148	687.798	15m54s	Completo
Duas requisições a serviços diferentes	2.071.978	9.950.000	11h58m17s	Parcial
Uma requisição com erro proposital	174.821	680.065	14m03s	Completo

Tabela 5.2: Dados obtidos no grupo B.

Terceiro grupo - Três serviços no ar

Assim como nos outros dois grupos, foi feita uma verificação sem requisição de serviço. O processo de verificação durou cerca de um minuto e quarenta e quatro segundos. Mais uma vez ficando evidente como o tempo necessário para a verificação cresce a cada linha

de execução acrescentada. Não foi encontrado *deadlock* ou qualquer outro problema real de implementação.

No segundo cenário foi feita uma requisição a um dos três serviços postos no ar. Já neste cenário, não foi possível gerar todo o espaço de estados do modelo. Todas as propriedades foram satisfeitas pelo modelo incompleto. Devido a não possibilidade de geração completa do espaço de estados nesse segundo cenário, os cenários com duas e três requisições feitas a serviços diferentes não foram verificados. Também não foi verificado o cenário com duas requisições a um mesmo serviço, pois foi verificado cenário equivalente no grupo A.

Foi verificado também um terceiro cenário. Igual ao segundo, exceto pela presença do erro inserido. O espaço de estados também não foi gerado completamente e as propriedades 1 e 2 também falharam, como nos dois primeiros grupos. A Tabela 5.3 mostra os dados obtidos nos experimentos realizados neste terceiro grupo.

Cenário	Estados	Transições	Tempo	Espaço de Estados
Sem requisição	28.887	96.805	1m44s	Completo
Uma requisição a um dos serviços	2.276.986	10.870.000	12h34m20s	Parcial
Uma requisição com erro proposital	2.200.816	9.702.000	11h14m40s	Parcial

Tabela 5.3: Dados obtidos no grupo C.

Considerações sobre as intercalações

Como pode ser notado, a adição de uma linha de execução aumenta bastante o espaço de estados e o tempo exigido para verificação. A diferença entre os tempos dos cenários é muito grande. Com invocações de serviços, a diferença cresce bastante, variando entre 17 segundos e pouco mais de 12 horas e meia (com espaço de estados parcial).

5.1.4 Resultados obtidos a partir de outras ferramentas

Esc/Java2

A princípio não foi possível verificar o *servidor de serviços* com esta ferramenta por que, como mencionado anteriormente no Capítulo 2, a versão 1.5 do SDK de Java não é suportada.

Foi instalada a versão 1.4 do SDK de Java mais estável para verificar o servidor com esta ferramenta. Alguns métodos foram anotados com pré- e pós-condições. A verificação ocorreu de forma rápida e alguns resultados foram interessantes, como possíveis valores nulos de certas variáveis. Além disso, algumas invariantes foram definidas para verificar se os serviços iniciados eram finalizados sem requisição do cliente. As invariantes foram obedecidas. Contudo, muitos avisos (um total de quarenta e sete) indicando falsos erros foram reportados, como os avisos a seguir.

```
ServiceDispatcher ...
ServiceDispatcher.java:37:
 Error:
 No such method _infixConcat_(java.lang.String,java.lang.String) in
  type java.lang.String System.out.println( "Mensagem: \"" + e.getMess ...
                                                                 ^
ServiceDispatcher.java:37:
 Error:
 No method valueOf(error) matching given argument types
  System.out.println( "Mensagem: \"" + e.getMess ...
```

Além dos avisos de “erros”, foram reportados três outros avisos. O primeiro e o segundo indicando possíveis referências nulas. E o terceiro indicando um possível índice fora dos limites de um determinado *array*.

```
Main: invokeService(ServerThread) ...
------------------------------------------------------------------------
Main.java:8: Warning: Possible null dereference (Null)
              while(!serverThread.serverIsUp);
                                  ^
Execution trace information:
 Reached top of loop after 0 iterations in "Main.java", line 8, col 2.
------------------------------------------------------------------------
[0.14 s 5034856 bytes]  failed
```

```
Main: main(java.lang.String[]) ...
------------------------------------------------------------------------
Main.java:14: Warning: Possible null dereference (Null)
               ServerThread thread = new ServerThread( services[0] ); ...
------------------------------------------------------------------------
Main.java:14: Warning: Array index possibly too large (IndexTooBig)
               ServerThread thread = new ServerThread( services[0] ); ...
------------------------------------------------------------------------
    [0.19 s 5311608 bytes]  failed

Main: Main() ...
    [0.04 s 5457480 bytes]  passed
  [0.921 s 5458208 bytes total]
3 warnings
```

Foi encontrado o erro inserido propositalmente no código. Entretanto, o erro presente na linha de execução principal dos serviços não foi encontrado. Vale a pena mencionar que a verificação só foi possível por que o sistema verificado não fazia uso de nenhum recurso da *API* 1.5 de Java.

Spex

Apesar de não ser realmente uma ferramenta, mas sim um projeto que estuda diferentes formas de especificação formal, o servidor de serviços foi verificado usando uma das abordagens usadas em Spex. Todo o código fonte foi traduzido para o modelo comportamental no formato da linguagem BIR. As classes e métodos pertencentes às *API*'s de terceiros também foram representadas por classes e métodos *stub*.

Apenas um cenário foi verificado: um serviço no ar com duas requisições a ele. Para que este cenário fosse verificado, foram necessários aproximadamente 45 minutos, com espaço de estados incompleto[2]. Os resultados obtidos foram praticamente os mesmos para as propriedades especificadas. Praticamente por que, apesar de os erros encontrados terem sido os mesmos, os contra-exemplos obtidos foram diferentes. Estes últimos foram mais detalhados devido à presença de mais variáveis do sistema no modelo, e devido também ao completo

[2] A verificação foi finalizada no momento em que foi detectado o erro

fluxo de execução do sistema. Isto mostra que aplicando um bom nível de abstração é possível obter os mesmos resultados em um tempo muito menor.

Jlint

O uso desta ferramenta na verificação do sistema usado neste estudo de caso gerou alguns resultados. Esta ferramenta realmente verificou o sistema muito rapidamente. Contudo, apenas três avisos foram reportados.

```
ServerThread.java:25:
  Local variable 'service' shadows component of class 'ServerThread'.
ServiceDispatcher.java:54:
  Index [-2147483648,2147483647] may be out of array bounds.
java\lang\String.java:1:
  equals() was overridden but not hashCode().
Verification completed: 3 reported messages.
```

Em relação ao primeiro aviso, Jlint reporta este tipo de mensagem quando o construtor de uma determinada classe usa um parâmetro com nome igual a algum campo desta classe e este campo não é referenciado explicitamente com **this**. Neste caso, há um campo chamado *service* do tipo *StreamConnectionNotifier* e há um parâmetro usado pelo construtor, também chamado *service*, só que do tipo *ServiceEntry*. Como o compilador Java consegue distinguir qual variável usar, baseado no tipo, esta mensagem não é tão relevante, pois não há erro no código.

O segundo aviso indica que pode ser usado um índice que esteja fora dos limites de um *array*. E o terceiro aviso diz respeito a uma classe da própria *API* de Java.

Apesar do processo de verificação ter sido muito rápido, o resultado não apresentou muita informação. Por não ser possível especificar as próprias propriedades, o desenvolvedor não é capaz de verificar suas próprias propriedades. Desta forma, muitos aspectos do sistema considerados relevantes podem ficar de fora da verificação.

Java PathFinder

A ferramenta *Java PathFinder* também não lê *bytecodes* gerados com compilador Java 1.5. Foi necessário mudar para SDK 1.4 e recompilar o código fonte.

Num primeiro momento, nenhuma propriedade foi especificada na verificação. O processo de verificação levou cerca de três minutos e trinta segundos. A ferramenta não detectou possibilidade alguma de *deadlock* ou quaisquer exceções não manipuladas. Também não foram detectadas possíveis referências nulas como nas demais ferramentas usadas nos experimentos.

Num segundo momento, foram especificadas propriedades em formas de asserções em alguns pontos do código. Asserções para garantir que nenhum dos serviços fosse finalizado sem que isso fosse mandado. Nenhuma das propriedades foi violada, exceto a propriedade referente ao bloqueio na linha de execução principal. Também foi detectada a referência nula causada pela inserção do erro no código. O processo de verificação, com propriedades especificadas, levou cerca de quinze minutos. Uma limitação encontrada no uso desta ferramenta foi a necessidade de se analisar cuidadosamente o *trace* produzido. Por ser muito grande, possuir muita informação detalhada, a detecção do problema apontado pela ferramenta demandou tempo e muita atenção.

5.2 Monitor de Energia

Nesta seção são apresentados os experimentos realizados com um sistema que monitora o nível de energia da bateria de dispositivos móveis. Como os experimentos realizados com o primeiro sistema, apresentado na Seção 5.1, foram feitos pelo autor da metodologia e linguagem de anotação definidos neste trabalho, fez-se necessária a realização de um experimento que fosse feito por um desenvolvedor sem vínculo algum com este trabalho, mas que estivesse disposto a usar e avaliar a linguagem de anotação JaCA.

5.2.1 Apresentação

No ano de 1991, Mark Weiser descreveu a base do que hoje é conhecido por *computação ubíqua*, ou *computação pervasiva* [Wei91]. Este tipo de computação está embutida nos objetos do dia a dia, como geladeiras, televisões e celulares, por exemplo. Estes objetos interagem entre si com o propósito de beneficiar o usuário.

É neste contexto que se insere este segundo estudo de caso. Trata-se de uma aplicação que monitora o nível de energia da bateria de determinado dispositivo. Esta aplicação foi

desenvolvida no trabalho apresentado por Loureiro em [LBB+06]. Na Figura 5.9 é ilustrado o comportamento do sistema. As aplicações (*App1*, *App2* e *App3*) executadas no dispositivo *Device* se cadastram na aplicação, chamada *Monitor*, que monitora a bateria. Para cada uma das aplicações cadastradas, o monitor da bateria cria uma nova linha de execução e associa a linha à aplicação cadastrada. Quando a bateria atinge determinado nível de energia considerado baixo (ou de risco), todas as aplicações interessadas no estado da bateria recebem um aviso de suas respectivas linhas de execução (*L1*, *L2* e *L3*) criadas pelo sistema *Monitor*.

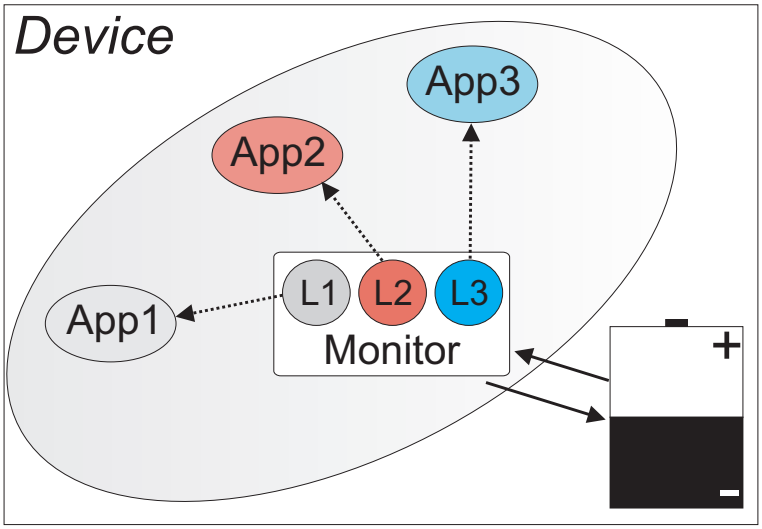

Figura 5.9: Sistema monitor de bateria.

Todas as anotações de código, comportamentais e de propriedades, são mostradas no Apêndice E.

Comportamentos desejáveis

Era desejado, pelo desenvolvedor, que apenas não houvesse *deadlock* no sistema modelado. Além disso, foram especificadas duas propriedades:

1. Em algum momento o nível de energia da bateria estará dentro da margem que é considerada baixa.

2. Chegando a um nível baixo, todas as aplicações cadastradas receberão a notificação de nível de energia baixo.

5.2.2 Resultados

Foram definidos três cenários para este sistema: um cenário com uma aplicação cadastrada, outro com duas aplicações cadastradas e um terceiro cenário com três aplicações cadastradas. As propriedades foram satisfeitas em todos os cenários definidos. Não houve *deadlock* e sempre que o nível de energia atingia um nível considerado baixo, as aplicações cadastradas eram notificadas. A Tabela 5.4 contém os dados obtidos a partir dos experimentos realizados. Mais uma vez pode ser visto como a intercalação entre as linhas de execução aumenta o tempo necessário à verificação do sistema.

Cenário	Estados	Transições	Tempo	Espaço de Estados
Uma aplicação	1.082	2.778	1s	Completo
Duas aplicações	482.094	2.138.030	25m23s	Completo
Três aplicações	4.875.950	26.379.999	35h21m00s	Parcial

Tabela 5.4: Dados obtidos nos experimentos do segundo estudo de caso.

Outro resultado interessante foi obtido através do contato direto do desenvolvedor, leigo em modelagem formal, com a linguagem de anotação. Foi visto que, mesmo para um desenvolvedor sem experiência em modelagem, a linguagem definida neste trabalho é assimilada de forma rápida. O contato inicial com a linguagem JaCA fez com que o programador expressasse todo o comportamento do código, encarando JaCA como mais uma linguagem de programação e replicando todo o código fonte. Após saber do ônus inerente à não abstração do código, a linguagem de anotação passou a ser usada corretamente, como linguagem de modelagem que tem como objetivo abstrair, e não replicar todo o comportamento.

Foi percebido que por se assemelhar muito com Java, as primeiras atividades de modelagem não serão suficientemente abstratas. Por não possuir experiência com verificação de modelos, o desenvolvedor acaba insistindo em replicar o seu código fonte. Tal problema pode ser contornado com a prática ou aviso prévio ao desenvolvedor das conseqüências causadas pela falta de abstração no modelo comportamental. No Apêndice D, é apresentado um pequeno relatório sobre esta experiência.

5.3 Conclusões

Os experimentos aqui apresentados foram bastante interessantes no que diz respeito não só à validação da técnica, mas também à validação da ferramenta e da linguagem de anotação aqui implementados. Apesar de alguns componentes ainda não estarem completamente implementados (e.g., tradutor de LTL para funções BIR), o *plugin* automatizou boa parte do processo de verificação do *servidor de serviços* e do *monitor de bateria*. Algumas falhas de implementação do *plugin* foram detectadas durante o processo de verificação do primeiro sistema apresentado. Além disso, os dois sistemas estudados nos experimentos serviram como teste para a ferramenta.

Foi detectado também que algumas porções de anotações da linguagem JaCA ainda não são traduzidas para BIR 100% de forma correta (e.g., chamadas de métodos aninhados). Desta forma, algumas alterações manuais foram necessárias nos modelos BIR para que as verificações fossem feitas. Alguns ajustes (e.g., nos componentes que reportam erros de anotação ao programador) ainda são necessários no protótipo para que ele fique completamente funcional.

Os processos de anotação dos dois sistemas apresentados foram realizados por uma única pessoa (uma pessoa em cada sistema). No caso de mais de uma pessoa participar deste processo, é necessário que haja um consenso sobre o que será ou não abstraído. Desta forma, modelos comportamentais inconsistentes são evitados.

A anotação comportamental de todo o código, do primeiro sistema, demandou cerca de duas horas e trinta e dois minutos para ser finalizada[3]. As anotações não foram feitas pelo autor da implementação do *servidor de serviços*. Por isso, foi necessário um tempo para se entender o código fonte, e assim modelá-lo. Caso as anotações tivessem sido feitas pelo autor do sistema, um tempo extra teria de ser empregado para que este aprendesse a linguagem de anotação. Uma vez conhecedor da linguagem de anotação, o desenvolvedor não precisaria gastar tempo estudando tal linguagem. Entretanto, um modelador precisaria sempre aprender a semântica do sistema a ser modelado.

As anotações de especificações de propriedades demandaram mais tempo, no primeiro estudo de caso. Foram necessárias algumas interações com o desenvolvedor para saber o

[3] Apenas as anotações comportamentais.

que ele realmente queria verificar no modelo. Após estas interações, a fase de especificação fez com que o modelador (não o desenvolvedor) conhecesse bastante o comportamento do código implementado pelo desenvolvedor. Assim, conclui-se que caso o próprio desenvolvedor especificasse suas próprias propriedades, o mesmo poderia ter aprendido ainda mais sobre sua implementação, facilitando a busca por possíveis erros.

Nos experimentos realizados com o segundo sistema apresentado, o processo total de anotação comportamental demorou cerca de uma hora. Somado ao tempo de quinze minutos para aprender a sintaxe da linguagem de anotação, o desenvolvedor levou cerca de uma hora e quinze minutos para anotar seu código. Apesar de menos anotações terem sido feitas neste segundo estudo de caso, comparando-se ao tempo gasto no primeiro estudo de caso (duas horas e trinta minutos), o tempo de anotação foi relativamente maior devido à falta de prática do desenvolvedor em modelagem de sistemas. Visto que foi sua primeira experiência com modelagem, é de se esperar que modelagens futuras demandem menos tempo. Além disso, apesar dos esforços realizados neste trabalho, fica muito difícil para o desenvolvedor anotar as propriedades desejadas, pois este geralmente não possui conhecimento em lógica temporal. No segundo sistema, as propriedades lógico-temporais foram especificadas pelo autor deste trabalho.

Além de terem sido encontrados um erro na implementação do sistema *servidor de serviços* e outro erro inserido de forma proposital, os experimentos com o primeiro sistema apresentado, juntamente com o segundo, serviram para se obter dados sobre quão onerosa esta técnica pode ser dentro de um processo de verificação de software, em se tratando de tempo empregado. Especificações mais abstratas demandam pouquíssimo tempo para serem verificadas. Contudo, muitos detalhes não podem ser verificados. Nos estudos aqui apresentados, algumas variáveis foram abstraídas, mas as etapas de verificação em si demoraram um tempo razoável devido às informações que não puderam ser abstraídas e às informações adicionais[4]. Cenários simples demandaram um segundo para suas verificações, enquanto a verificação de outros mais complexos chegou a durar mais de doze horas.

O tempo necessário para se verificar as propriedades junto ao modelo demandou um tempo um pouco maior, cerca de trinta por cento a mais, o que pode servir como parâmetro

[4]Alguma variáveis booleanas globais adicionadas ao modelo para que fosse possível verificar ordem de chamadas de métodos.

a ser analisado sobre o que se deve verificar mais detalhadamente.

No servidor de serviços foram anotados 41 métodos e um total de 13 classes, sendo 10 métodos *stubs* e 7 classes *stubs*. O número total de linhas de código fonte e de anotações foi de 352 e 192, respectivamente. Já no sistema para monitoração de bateria, o programador anotou 14 métodos e um total de 10 classes, sendo 2 métodos *stubs* e 1 classe *stub*. O número total de linhas de código fonte e de anotações para o monitor de bateria foi de 127 e 63, respectivamente.

Tratando-se de detecção de falhas, o trabalho aqui apresentado se mostra mais completo, em relação aos requisitos mencionados na Seção 2.2.3, do que as ferramentas Jlint e Perfect Developer. Pois é capaz de verificar propriedades desejadas em programas concorrentes. Em relação a todos os trabalhos relacionados apresentados, o diferencial é que neste aqui, o usuário possui pleno controle sobre a abstração aplicada. Por meio de uma linguagem semelhante a uma linguagem de programação, o próprio desenvolvedor modela seu sistema. Este exercício de modelagem proporciona ao usuário um conhecimento maior sobre o comportamento de seu próprio código. Contudo, uma limitação deste trabalho é como as propriedades são especificadas. Apesar de ser possível usar asserções, as propriedades mais expressivas são descritas usando a lógica temporal LTL, o que representa uma certa limitação dado que poucos são conhecedores desta linguagem de especificação.

Capítulo 6

Considerações Finais

O verificador de modelos Bogor e a linguagem de modelagem BIR foram essenciais para este trabalho. Graças ao suporte oferecido por eles à linguagem orientada a objetos e à concorrência, é possível modelar sistemas concorrentes orientados a objetos sem que se seja necessário fazer uso de artifícios específicos da linguagem formal que sejam semanticamente diferentes dos recursos deste tipo de linguagem.

Um bom aspecto do verificador de modelos é que há uma comunidade de pesquisadores relativamente grande e ativa utilizando o programa em seus trabalhos. Há um fórum de discussões sobre o uso da ferramenta onde experiências (dúvidas, relatos, etc) são compartilhadas entre os membros desta comunidade. Desta forma, a cada dia que passa, um *bug* é descoberto e rapidamente é eliminado do software. Dois bons exemplos disto foram:

- *invocação de métodos* - numa das primeiras versões, não era possível fazer invocação de métodos na representação de alto nível da linguagem BIR, o que foi providenciado rapidamente a retificação pela equipe de desenvolvimento do verificador de modelos.

- *sentença **choose*** - foi detectado um erro na geração de contra-exemplos em sentenças *choose*. Anteriormente, mesmo com mais de uma opção avaliada em *true*, o verificador gerava caminho no espaço de estados apenas para a primeira opção *when*, desprezado as demais. Foi reportado no fórum este erro e pouco tempo depois ele foi corrigido.

Apesar desta evolução rápida do verificador de modelos, a extensão responsável pela verificação de propriedades descritas em LTL ainda não está funcionando, apesar de constar

na documentação que sim. Para contornar este problema, uma solução foi adotada. Abaixo são descritos os passos desta solução:

1. traduzir cada uma das fórmulas LTL para a forma *never claim*, ou seja, negar a fórmula inteira e trazer a negação para junto das proposições;

2. utilizar o verificador de modelos Spin [SPI] para transformar as especificações do formato *never claim* para formato FSA (*Finite State Automaton*);

3. adicionar algumas informações (extensões) do Bogor aos autômatos (FSA);

4. e finalmente, verificar usando uma outra extensão do Bogor: *verificação de especificações de propriedades em FSA*.

Mesmo assim, algumas propriedades não podem ser efetivamente verificadas. Além disso, o tempo gasto na verificação do modelo com propriedades é bem maior do que na verificação padrão, sem propriedades especificadas.

Em relação à metodologia aqui desenvolvida, ela se mostrou bastante eficiente. A capacidade de o desenvolvedor abstrair o que considera irrelevante para a verificação desejada ajuda a diminuir o tempo necessário para a verificação do sistema, ou de parte dele. Contudo, como no uso de qualquer outra linguagem de modelagem, a medida certa a ser usada para que se possa obter alguns resultados sem que haja perda de resultados considerados relevantes é atingida com a prática na modelagem.

Outras técnicas, que não fazem uso de anotação, requerem muitas alterações no software para que seja gerado um modelo formal diretamente a partir do código fonte. Além disso, estes trabalhos requerem que o sistema esteja completamente implementado.

Uma característica fundamental e importante desta técnica, para que seu uso por desenvolvedores de uma forma geral seja possível, é que a linguagem de descrição comportamental (definida neste trabalho) usada para modelar o código fonte, é bastante semelhante a linguagem de programação Java. Desta forma, qualquer desenvolvedor, conhecedor ou não de alguma linguagem de modelagem formal, pode fazer uso da técnica de verificação de modelos. Contudo, os experimentos realizados com o sistema de monitoração de energia, apresentado na Seção 5.2, mostraram que em virtude da semelhança com a linguagem de programação, o desenvolvedor tende a replicar o código implementado.

Além disso, a atividade de modelar o próprio código faz com que o desenvolvedor exercite a forma como ele enxerga o programa. O desenvolvedor passa a conhecer melhor o seu próprio código, passa a conhecer melhor o comportamento do mesmo.

A ferramenta apresentada no Capítulo 4 é importante pois automatiza todo o processo da técnica apresentada neste trabalho. Basta ao usuário anotar o seu código com a linguagem de descrição comportamental definida neste trabalho e apresentada no Capítulo 3.

Além disso, por se tratar de uma ferramenta no formato de *plugin* para o ambiente de desenvolvimento Eclipse, ela se insere mais facilmente no processo de desenvolvimento. Por se integrar a IDE de desenvolvimento (Eclipse), o usuário deixa de ter que gerenciar as etapas de desenvolvimento e modelagem/verificação em aplicativos diferentes. Estas etapas ficam mais próximas uma da outra graças à IDE única.

Apesar da falha do verificador de modelos Bogor mencionada anteriormente, relacionada à verificação de propriedades LTL, é pretendido preparar o *plugin* para esta funcionalidade, implementando o componente de tradução de LTL para funções em BIR.

Baseado no *feedback* gerado pela experiência com o desenvolvedor, leigo em modelagem formal, obteve-se o forte indício de que a linguagem de anotação aqui definida cumpriu seu papel: servir como linguagem de descrição comportamental, que fosse de fácil uso para programadores, para a modelagem de programas Java concorrentes.

Apesar de não ter sido formalizada, a sintaxe e semântica desta linguagem estão bem definidas e há um mapeamento direto de qualquer modelo descrito usando a linguagem de anotação JaCA para um modelo comportamental BIR, que é uma linguagem de modelagem formalizada.

De uma forma geral, o trabalho aqui apresentado torna o uso da técnica de verificação de modelos mais viável no processo de desenvolvimento de software de uma forma geral. Isto por que a linguagem de descrição comportamental é flexível o suficiente para que o desenvolvedor possa aplicar, em seu modelo, o nível de abstração que considerar necessário para uma verificação eficiente. Mesmo aplicando um nível de abstração muito grande, o software terá sido verificado formalmente. Software este, que dificilmente teria sido submetido a qualquer tipo de verificação, muito mesmo uma formal.

Este trabalho possibilitou uma interação curta, porém interessante, com um outro grupo de pesquisa, chamado *Bughunters Extraordinare Verification and Validation Laboratory*

(`http://vv.cs.byu.edu/index.html`). Houve troca de informações sobre experimentos de ambos os grupos, proporcionando uma ajuda mútua.

Além disso, algumas falhas detectadas no verificador de modelos aqui usado, bem como algumas "soluções" para estas falhas, foram reportadas diretamente para um dos membros da equipe desenvolvedora do verificador Bogor. Isto se deu graças à interação com o grupo de pesquisa mencionado acima.

Alguns trabalhos podem ser começados a partir deste ponto. Antes disso, faz-se necessário finalizar a implementação do *plugin*. Alguns pequenos ajustes devem ser feitos para eliminar os erros existentes, como nos componentes que reportam erros ao programador. Outra tarefa interessante seria automatizar o processo de conversão das fórmulas LTL em *never claims* (equivalente aos autômatos de *Büchi*), pois este processo foi feito de forma totalmente manual.

Um trabalho interessante a ser feito, seria a geração de casos de teste a partir das anotações feitas junto ao código. Além de servir como modelo comportamental do programa, algumas anotações, como asserções (*assert*), poderiam servir na geração automática de casos de teste.

Uma integração com um dos trabalhos de mestrado do mesmo grupo de pesquisa pode ser realizada. No trabalho apresentado em [SM05], há geração de objetivos de teste baseado na técnica de verificação de modelos. Os dois trabalhos poderiam ser unificados numa única ferramenta em que seriam verificadas as propriedades especificadas em lógica temporal junto ao modelo do programa, e a partir destas fórmulas e dos contra-exemplos gerados, seriam gerados objetivos de teste para serem executados junto à implementação.

Para amenizar o problema da replicação do código implementado nas anotações, um tutorial de boas práticas de modelagem, usando a linguagem JaCA, poderia ser feito para que programadores possam obter modelos mais abstratos sem que percam informações relevantes.

Em relação às anotações, as anotações ao nível de classe poderiam ser resumidas a uma simples *tag* que indicasse a presença ou não da classe no modelo a ser gerado. Uma nova anotação, ao nível de atributo, poderia ser criada para indicar quais atributos de certa classe seriam introduzidos no modelo. Desta forma, o desenvolvedor não teria que redefinir os atributos da forma que é feita.

E por último, para facilitar ainda mais o uso da técnica aqui apresentada, a criação de uma linguagem de especificação de propriedades mais amigável poderia ser feita para “substituir’ a linguagem LTL. Esta nova linguagem a ser definida teria um poder de expressividade menor do que LTL. Entretanto, por estar mais próxima da linguagem natural (sintaticamente), seria mais compreensível aos desenvolvedores totalmente leigos em métodos formais. Esta linguagem poderia ser definida baseada nos padrões de propriedades apresentados em [DAC99].

Apêndice A

Verificação de Modelos

Neste apêndice é apresentada a técnica de verificação de modelos de uma forma geral, mostrando suas etapas e como a mesma funciona. São apresentados os dois tipos de lógica temporal utilizados para especificação de propriedades. Numa segunda parte, é apresentada uma comparação entre o poder de especificação das duas. E por último, são apresentadas algumas conclusões sobre o apêndice.

A técnica

A verificação de modelos é uma técnica automática para analisar o espaço de estados finito de sistemas concorrentes [CWA+96]. Na Figura A.1, é mostrado o esquema da técnica de verificação de modelos. Sua aplicação ocorre tradicionalmente através de três etapas:

1. Modelagem: esta etapa consiste em construir um modelo formal do sistema, fazendo uso de alguma linguagem formal, e a partir dele, obter *todo* o comportamento possível do sistema. O modelo que contém todo o comportamento obtido a partir do modelo formal é conhecido como *espaço de estados* do sistema.

2. Especificação: esta etapa consiste em especificar as propriedades comportamentais *desejáveis* do sistema. Um comportamento que se deseja do sistema pode ser descrito formalmente através de lógicas temporais ou máquinas de estado.

3. Verificação: esta etapa consiste em verificar se as especificações escritas são satisfeitas pelo modelo (espaço de estados). Esta etapa é feita de forma automática pela ferra-

menta chamada *verificador de modelos*. Esta ferramenta produz como resultado um valor verdade que indica se a especificação é satisfeita ou não pelo modelo. Em caso negativo, o verificador retorna ao usuário uma seqüência de estados de uma determinada execução, chamada de *contra-exemplo*, que demonstra que a especificação não é válida no modelo.

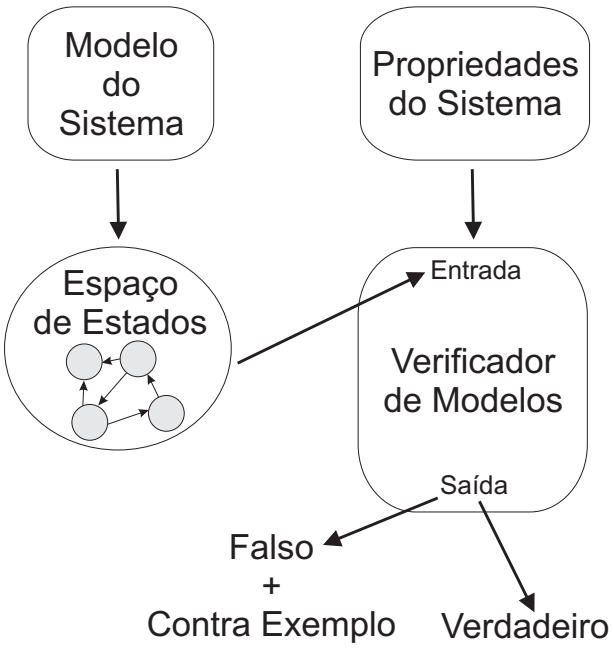

Figura A.1: Esquema da técnica de verificação de modelos.

A verificação de modelos pode ser aplicada em sistemas reativos ou não. Sistemas de natureza reativa tipicamente recebem estímulos do ambiente no qual estão inseridos e quase que imediatamente reagem a estes estímulos recebidos. Tradicionalmente, eles podem ser complexos, distribuídos, concorrentes e podem não possuir um término de execução, isto é, eles podem estar constantemente em execução, prontos para interagir com o usuário ou outros sistemas, reativos ou não. Este conjunto de características exige que as propriedades destes sistemas sejam definidas não apenas em função de valores de entrada e saída, mas também em relação à *ordem* em que os eventos ocorrem.

As lógicas temporais são utilizadas para a especificação de propriedades em verificação de modelos porque elas são capazes de expressar relações de ordem, sem recorrer à noção explícita de tempo. Tradicionalmente, duas formalizações de lógica temporal são utilizadas no contexto de verificação de modelos: LTL (*Linear Temporal Logic*) [Pnu77] e CTL (*Compu-*

tation Tree Logic) [Sif90]. A abordagem em LTL, conhecida também como lógica temporal linear, considera que uma propriedade pode ser quantificada para todas as execuções do sistema. A abordagem em CTL, também conhecida como lógica temporal ramificada, por sua vez, considera que uma propriedade pode ser quantificada para uma ou para todas as execuções do sistema.

O principal desafio à aplicação de verificação de modelos em situações reais é o famoso problema conhecido como *explosão do espaço de estados*. Armazenar todos os comportamentos possíveis de um sistema, mesmo sendo um sistema simples, pode esgotar os recursos de memória de uma máquina, mesmo que o número de estados alcançados pelo sistema seja finito. Muitos trabalhos têm sido desenvolvidos neste contexto e há, atualmente, um número considerável de técnicas para tratar deste problema. Na área de desenvolvimento de sistemas de hardware, por exemplo, a técnica para a representação simbólica do espaço de estados, desenvolvida por McMillan, viabilizou a aplicação de verificação de modelos no protocolo IEEE Futurebus+ [CGH+93]. Na área de desenvolvimento de sistemas de software, a explosão do espaço de estados constitui uma barreira ainda maior à aplicação de verificação de modelos, por isso, a representação simbólica nem sempre pode ser adequadamente aplicada, pois sistemas de software são mais complexos em sua natureza, com um comportamento não tão limitado como nos sistemas de hardware. Desta forma, outros trabalhos para tratar deste problema foram desenvolvidos: interpretação abstrata [DHJ+01b], redução de ordem parcial [GPS96; Val98], entre outros.

Lógica Temporal Linear

Sintaxe e semântica A sintaxe da lógica temporal linear (LTL) tem como ponto de partida o conjunto de proposições atômicas, denotado por *AP*. Uma proposição atômica *p* é uma sentença que informa algo a respeito de um determinado estado do sistema, ela pode ser interpretada como sendo verdadeira ou falsa. Algumas proposições atômicas podem ser: “x é igual a zero”, “o objeto *obj* é diferente de nulo”, “a linha de execução *thread1* está em execução”, etc. A definição seguinte determina o conjunto de fórmulas que pode ser declarado em lógica temporal linear.

Definição A.1 (Sintaxe de LTL) *Seja AP o conjunto de proposições atômicas, então*

1. *cada proposição $p \in AP$ é uma fórmula LTL;*

2. *se ϕ e ψ são fórmulas LTL, então $\neg\phi$, $\phi \wedge \psi$, $X\phi$, $F\phi$, $G\phi$ e $\phi U \psi$ também são.*

Os operadores temporais são X (próximo), G (globalmente), F (futuramente) e U (até). Há alguns outros operadores, mas por hora, apenas estes serão utilizados. Formalmente, a lógica temporal linear é interpretada em uma seqüência de estados. Intuitivamente, $X\phi$ significa que a fórmula ϕ deve valer (é verdadeira) no próximo estado da seqüência, $\phi U \psi$ significa que ϕ deve valer até que ψ seja verdadeira. $G\phi$ significa que ϕ deve ser verdadeira no estado atual e em todos estados alcançáveis, em outras palavras, deve ser válida sempre a partir do estado atual. $F\phi$ significa que ϕ deve ser verdadeira no estado atual ou em algum estado do futuro, alcançável a partir daquele estado. Os operadores temporais G (globalmente ou sempre) e F (futuramente) possuem certas relações de equivalência e podem ser definidos por:

$$G\phi \equiv \neg F \neg \phi$$

e

$$F\phi \equiv trueU\phi$$

As noções de "estado" e de "próximo estado" são definidas em relação à *estrutura de Kripke*. Esta estrutura é apresentada na seguinte definição:

Definição A.2 (Estrutura de Kripke) *Uma estrutura de Kripke é uma tupla $\mathcal{M} = (S, I, R, Label)$ na qual:*

- *S é um conjunto finito de estados,*

- *$I \subseteq S$ é um conjunto de estados iniciais,*

- *$R \subseteq S \times S$ é uma relação de transição satisfazendo*

$$\forall s \in S.(\exists s' \in S.(s, s') \in R)$$

- *$Label : S \rightarrow 2^{AP}$, associando a cada estado s de S, proposições atômicas $Label(s)$ que são válidas em s.*

Em outras palavras, uma estrutura de Kripke é uma máquina de estados finita que representa todas as possíveis execuções de um sistema, ou como na maioria das vezes, todas as possíveis execuções do modelo de um sistema. Cada estado do sistema é rotulado com as proposições atômicas que são verdadeiras nele. Segundo a definição de R, cada estado deve possuir ao menos um sucessor. Desta forma, situações reais nas quais um estado s não possui um sucessor (*deadlock*) devem ser representadas através de $(s, s) \in R$, isto é, através de auto-laço.

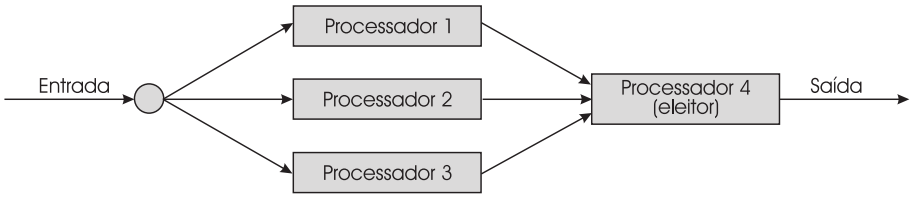

Figura A.2: Sistema redundante modular triplo.

Para exemplificar uma estrutura de Kripke, considere o sistema tolerante a falhas, ilustrado na Figura A.2, formado por três processadores que geram resultados para um quarto que é capaz de eleger qual resposta utilizar. Inicialmente todos os processadores estão funcionando, porém todos estão sujeitos a falhas durante a execução. De forma simples, o estado $S_{i,j}$ representa (ou modela) que i processadores produtores de resultados ($0 \leq i < 4$) e j processadores eleitores ($0 \leq j \leq 1$) estão em funcionamento. Quando um processador falha, ele pode ser reparado e logo em seguida voltar a funcionar. É considerado que apenas um componente pode ser reparado por vez. Quando o processador eleitor falha, todo o sistema pára de funcionar. O conjunto de proposições atômicas deste problema é $AP = \{up_i | 0 \leq i < 4\} \cup \{down\}$. A proposição up_0 denota que somente o processador eleitor está operacional, up_1 denota que além do processador eleitor, um outro também está operacional e assim por diante. A proposição $down$ informa que nenhum dos processadores do sistema está funcionando. A estrutura de Kripke para este sistema é definida da seguinte

forma:

$$
\begin{aligned}
S &= \{S_{i,1} |\ 0 \leq i < 4\} \cup \{S_{0,0}\} \\
I &= \{S_{3,1}\} \\
R &= \{(S_{i,1}, S_{0,0} |\ 0 \leq i < 4)\} \cup \{(S_{0,0}, S_{3,1})\} \cup \\
&\quad \{(S_{i,1}, S_{i,1} |\ 0 \leq i < 4)\} \cup \{(S_{i,1}, S_{i+1,1} |\ 0 \leq i < 3)\} \\
&\quad \cup \{(S_{i+1,1}, S_{i,1} |\ 0 \leq i < 3)\} \\
Label(S_{0,0}) &= \{down\}\ e \\
Label(S_{i,1}) &= \{up_i\}\ para\ 0 \leq i < 4
\end{aligned}
$$

A representação gráfica da estrutura de Kripke para esse sistema é ilustrado na Figura A.3.

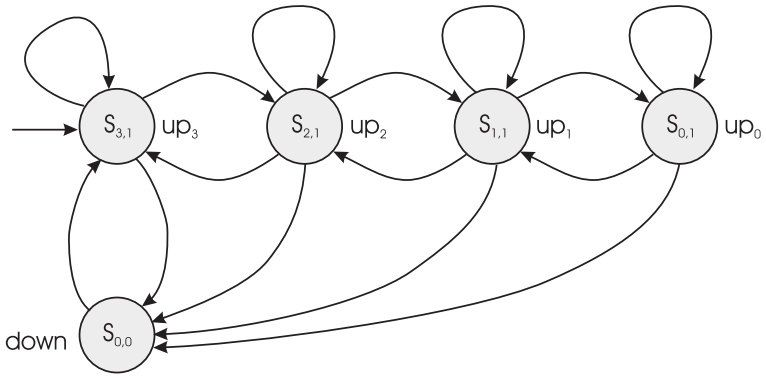

Figura A.3: Estrutura de Kripke para o *sistema redundante modular triplo*.

Para se definir formalmente a semântica de LTL, o conceito formal de caminho é apresentado pela seguinte definição:

Definição A.3 (Caminho) *Um caminho em $\mathcal{M}$ é uma seqüência infinita de estados $s_0,\ s_1,\ s_2,\ \ldots$ tal que $s_0 \in I$ e $(s_i, s_{i+1} \in R$ para todo $i \geq 0)$.*

Portanto, um caminho é uma seqüência infinita de estados que representa uma possível execução do sistema a partir do seu estado inicial. $\sigma[i]$ denota o (i+1)-ésimo estado de σ e σ^i representa o sufixo de σ obtido pela remoção do(s) i-primeiro(s) estados de σ. A função $Caminhos(s)$ determina todos os possíveis caminhos da estrutura $\mathcal{M}$ que tem como estado inicial o estado s.

Uma vez definida a estrutura na qual LTL é interpretada, a definição a seguir apresenta formalmente a linguagem LTL.

Definição A.4 (Semântica de LTL) *Sejam $p \in AP$ uma proposição atômica, σ caminho infinito e ϕ, ψ fórmulas LTL, a relação "satisfaz", denotada por $\models$, é definida por:*

$\sigma \models p \quad \Leftrightarrow p \in Label(\sigma[0])$

$\sigma \models \neg\phi \quad \Leftrightarrow (\sigma \not\models \phi)$

$\sigma \models \phi \wedge \psi \quad \Leftrightarrow (\sigma \models \phi)$ **e** $(\sigma \models \psi)$

$\sigma \models X\phi \quad \Leftrightarrow \sigma^1 \models \phi$

$\sigma \models \phi U \psi \quad \Leftrightarrow \exists j \geq 0, (\sigma^j \models \psi$ **e** $(\forall 0 \leq k < j, \sigma^k \models \phi))$

Apresentada a formalização da lógica LTL, o problema da verificação de modelos é definido formalmente da seguinte maneira:

Definição A.5 (Verificação de modelos usando LTL) *Dado o modelo $\mathcal{M}$ formalmente representado pela estrutura de Kripke $\mathcal{M} = (S, I, R, Label)$ e uma fórmula LTL ϕ:*

$$\mathcal{M} \models \phi \; se \; e \; somente \; se \; \forall s \in I, (\forall \, Caminhos(s), \sigma \models \phi)$$

Em outras palavras, esta definição significa que a propriedade ϕ vale no modelo $\mathcal{M}$ se e somente se ϕ vale para todos os caminhos que se iniciam a partir de algum estado inicial. Em termos de um sistema isto quer dizer que para uma propriedade ser verificada, ela obrigatoriamente tem que valer para todas as possíveis execuções do sistema.

Algumas possíveis propriedades, baseadas nas proposições atômicas apresentadas anteriormente, do sistema apresentado na Figura A.2, poderiam ser: *F(down)*, *G(down* $\rightarrow$ *F(up$_3$))* ou *G((up$_0$*$\vee$ *up$_1$*$\vee$ *up$_2$*$\vee$ *up$_3$)* $\rightarrow$ *F(down))*. Dentre estas três propriedades, apenas a segunda é satisfeita pelo modelo. As outras duas não são satisfeitas por que existem execuções do modelo que não as satisfazem.

A primeira propriedade significa que, *num momento futuro o sistema parará de funcionar*. Isto pode não acontecer, pois um caminho possível dentro da estrutura de Kripke (ou espaço de estados) deste sistema pode ser *up$_3$up$_2$up$_3$up$_2$...*, ou simplesmente *(up$_3$up$_2$)^w*.

A segunda propriedade significa que, *sempre que o sistema não estiver funcionando, em algum momento no futuro ele voltará a funcionar com todos os seus processadores*. Esta propriedade é fácil de ser notada, dado que a partir do estado *down*, todos os caminhos levam

ao estado up_3. Esta propriedade poderia ter sido descrita pela seguinte fórmula equivalente: $G(down \rightarrow X(up_3))$ [1].

E a terceira propriedade significa que, *sempre que o sistema estiver funcionando com o processador eleitor, juntamente com quaisquer dos processadores produtores ou nenhum destes últimos, em algum momento no futuro ele virá a parar de funcionar.* Esta propriedade é falsa do mesmo modo que a primeira o é.

Na primeira propriedade, o sistema pode ficar alternando infinitamente entre os estados up_3 e up_2. Já na terceira propriedade, o sistema pode simplesmente ficar alternando entre os estados de funcionamento eternamente sem que pare de funcionar. Este tipo de comportamento é muito improvável de acontecer mas não impossível, dado a estrutura de Kripke. Tais situações, conhecidas como *injustas* (*unfairs*), são irreais na prática e segundo [Kat99], elas devem ser evitadas.

Um exemplo de situação injusta pode ser visto no famoso problema do jantar dos filósofos, usado como objeto de experimentos em muitos trabalhos, dentre eles [Dij71]. Neste problema, um processo (filósofo) pode ficar para sempre em execução enquanto os outros ficam esperando por sua vez. Para evitar este problema, restrições de justiça (ou *fairness*) podem ser expressas como parte da especificação de propriedades. O formato geral de uma especificação de propriedade contendo restrição de *fairness* é:

$$restrição\ de\ fairness \rightarrow propriedade\ desejada$$

Podem ser usadas três diferentes formas de *fairness*: incondicional, fraca e forte. Para entender cada uma delas, considere Ψ como sendo a propriedade desejada, tal como ausência de *starvation*, e Φ a restrição de *fairness*, como por exemplo *um processo tem que ter sua vez regularmente*.

***Fairness* incondicional** Também conhecida na literatura como *imparcialidade*. Um caminho é incondicionalmente justo com respeito a Ψ se satisfaz $GF\Psi$. Se por acaso Ψ significar *um processo entra em sua região crítica* $\vee$ *um processo tem sua vez para ser executado*, então o caminho é incondicionalmente justo com respeito a estas propriedades se elas são

[1] Esta equivalência pode tranqüilamente não ocorrer em outras estruturas de Kripke. O fato de uma propriedade ocorrer num certo momento no futuro não significa necessariamente que ela deva ocorrer logo no próximo estado.

válidas vez ou outra no decorrer do caminho para sempre. Ou a primeira propriedade ou a segunda propriedade atende a esta condição. Caso isto aconteça, é dito que o *fairness* é incondicional. Reformulando GFΨ para que se fique mais claro a incondicionalidade, obtém-se *true* $\rightarrow$ *GF*Ψ.

***Fairness* fraca** Também conhecida na literatura como *justiça (justice)*. Um caminho é fracamente justo com respeito a Ψ e a restrição de justiça Φ se ele satisfaz: FG$\Phi \rightarrow$ GFΨ. Dado que Φ signifique *habilitado(a)* e Ψ signifique *executado(a)*, sendo a uma atividade, justiça fraca significa que tal atividade, uma transição ou um processo inteiro, está continuamente habilitada (FG habilitado(a)), então ela tem que ser executada vez ou outra no decorrer daquele caminho para sempre (GF executado(a)).

***Fairness* forte** Também conhecida na literatura como *compaixão (compassion)*. Um caminho é fortemente justo com respeito a Ψ e a restrição de justiça Φ se ele satisfaz: GF$\Phi \rightarrow$ GFΨ. A diferença para a justiça fraca é que a premissa FGΦ é trocada por GFΦ. Justiça forte significa que se uma atividade a está habilitada continuamente, mas não necessariamente sempre, pois pode haver períodos onde Φ não seja válida, ela (a atividade) será executada vez ou outra no decorrer daquele caminho para sempre.

Lógica Temporal Ramificada

Sintaxe e semântica A lógica temporal linear considera que uma formula é verdadeira num determinado estado se ela vale para todas as possíveis execuções a partir daquele estado. Em meados dos anos 80, Clarke e Emerson [CE81] propuseram uma lógica capaz de considerar diferentes futuros possíveis, através da noção de tempo ramificado. A idéia desta lógica é quantificar as possíveis execuções de um programa através da noção de caminhos que existem no espaço de estados do sistema. Agora as propriedades podem ser avaliadas em relação a todas as execuções ou então em relação a alguma execução. Esta lógica é utilizada em alguns verificadores de modelos, como é o caso do verificador de modelos SMV, apresentado em [McM93]. A sintaxe de CTL é formalmente dada pela seguinte definição:

Definição A.6 (Sintaxe de CTL) *Seja AP o conjunto de proposições atômicas, então:*

1. *cada proposição* $p \in AP$ *é uma fórmula CTL;*

2. *se* ϕ *e* ψ *são fórmulas CTL, então* $\neg\phi$, $\phi \wedge \psi$, $EX\phi$, $AX\phi$, $E[\phi U \psi]$, $A[\phi U \psi]$, $EG\phi$, $AG\phi$, $EF\phi$ *e* $AF\phi$ *também são.*

Informalmente, $EX\phi$ significa que ϕ deve valer em pelo menos um dos estados seguintes ao estado atual. Já $AX\phi$, significa que ϕ deve valer em todos os estados seguintes ao estado atual. A fórmula $E[\phi U \psi]$ significa que, para algum caminho que se inicia no estado atual, ϕ deve valer até que ψ seja verdadeiro. Já $A[\phi U \psi]$, significa que ϕ deve valer até que ψ seja verdadeiro em todos os caminhos que se iniciam no estado atual.

É fácil notar que o quantificador existencial (E) requer que a propriedade seja válida em pelo menos um dos caminhos que se iniciam no estado atual. Já o quantificador universal (A), requer que a propriedade seja válida em todos os caminhos que se iniciam a partir do estado inicial. $EG\phi$ ($AG\phi$) significa que, para algum (todo) caminho que se inicia no estado atual, ϕ deve valer sempre. $EF\phi$ ($AF\phi$) significa que, para algum (todo) caminho que se inicia no estado atual, ϕ deve valer no estado inicial ou em algum estado futuro.

Há algumas relações de equivalência entre os operadores. Cada um deles pode ser descrito através dos operadores $EX\phi$, $EG\phi$ e $E[\phi U \psi]$:

- $AX\phi \equiv \neg EX\neg\phi$;
- $EF\phi \equiv E[trueU\phi]$;
- $AG\phi \equiv \neg EF\neg\phi$;
- $AF\phi \equiv \neg EG\neg\phi$;
- $A[\phi U \psi] \equiv \neg E[\neg\psi U(\neg\phi \wedge \neg\psi)] \wedge \neg EG\neg\psi$;

Semanticamente, CTL pode ser definido sobre o mesmo modelo dado pela Definição A.2, ou seja, através de uma estrutura de Kripke. Porém a relação "satisfaz" agora considera os quantificadores de caminho existencial e universal.

Definição A.7 (Semântica de CTL) *Sejam* AP *o conjunto das proposições atômicas,* $p \in AP$*, o modelo* $\mathcal{M}$ *formalmente dado pela estrutura de Kripke* $\mathcal{M} = (S, R, I, Label)$,

$s \in S$ e ϕ e ψ fórmulas CTL. A relação satisfaz, denotada por $\models$, é definida por:

$s_0 \models p \quad \Leftrightarrow p \in Label(s_0)$

$s_0 \models \neg\phi \quad \Leftrightarrow s_0 \not\models \phi$

$s_0 \models \phi \wedge \psi \quad \Leftrightarrow (s_0 \models \phi)\ e\ (s_0 \models \psi)$

$s_0 \models EX\phi \quad \Leftrightarrow$ *para algum estado* t *tal que* $(s_0, t) \in R, t \models \phi$

$s_0 \models E[\phi U \psi] \quad \Leftrightarrow \exists\sigma \in Caminhos(s_0),\ (\exists j \geq 0, (\sigma^j \models \psi \wedge (\forall 0 \leq k < j,\ \sigma^k \models \phi))$

$s_0 \models EG\phi \quad \Leftrightarrow\ \exists\sigma\ \in\ Caminhos(s_0)$ *tal que* $\sigma^0, \sigma^1, ..., \sigma^n, \sigma^k,\ 0 \leq k \leq n \wedge (\sigma^n, \sigma^k) \in R, \forall i[0 \leq i \leq n, \sigma^i \models \phi]$

Na Figura A.4 é apresentada uma estrutura de Kripke e a verificação de algumas fórmulas em CTL. Os estados pintados de preto denotam que a fórmula é válida neles.

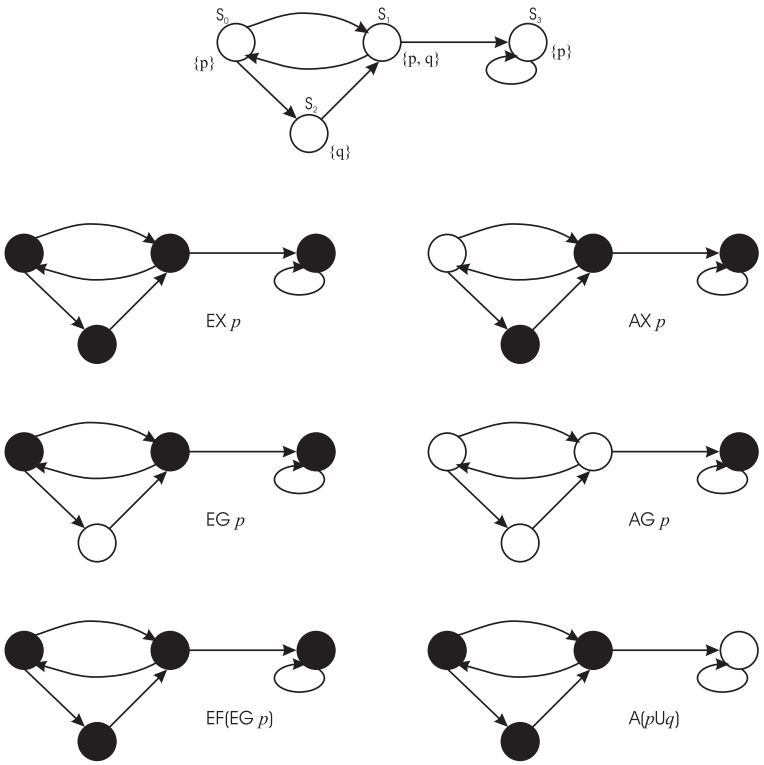

Figura A.4: Exemplo de estrutura de Kripke e verificação de fórmulas em CTL.

Os estados que não estão pintados de preto são aqueles onde a fórmula não vale. A fórmula EXp, que significa que existe pelo menos um estado, onde a proposição p é verdadeira, que é o próximo do estado atual, é válida em todos os estado da estrutura de Kripke. Tomando qualquer estado como estado inicial, esta fórmula será satisfeita pelo modelo. Já a fórmula EGp, que significa que existe pelo menos um caminho, iniciado no estado atual,

onde p será válido em todos os estados. Esta fórmula não é válida no estado S_2, onde apenas a proposição q é válida. Isto por si faz com que todos os caminhos que contém este estado, dentre eles os que começam pelo estado, não satisfaçam a propriedade.

Poder de expressividade de LTL e de CTL

Com a inclusão do quantificador existencial (E), alguém poderia pensar que CTL tem um poder de expressividade maior do que LTL. Poderia pensar inclusive que o poder de expressividade de LTL fosse um subconjunto do poder de expressividade de CTL. Porém isto não é verdade. As expressividades de LTL e CTL são incomparáveis. Na Figura A.5, é ilustrada a relação existente entre os poderes de expressividade de LTL e de CTL.

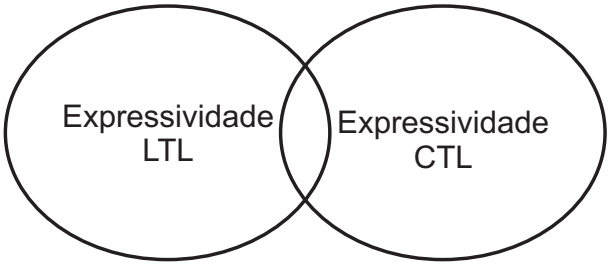

Figura A.5: Relação existente entre as expressividades de LTL e CTL.

Como ilustrado na Figura A.5, há propriedades que podem ser descritas em LTL e não podem ser descritas em CTL, e vice-versa. Verificadores de modelos, que fazem uso de LTL, trabalham em conjuntos de caminhos computacionais, conjuntos estes nos quais estão todos os caminhos computacionais individuais possíveis de um certo modelo. Desta forma, propriedades existenciais, como é o caso da fórmula AGEFϕ (*a partir de todos os estados existe um caminho onde ϕ é válido*), não podem ser expressadas. A capacidade de se expressar propriedades existenciais é bastante útil na busca de possíveis *deadlocks* num *design* de sistema.

Por outro lado, CTL não é capaz de expressar certas propriedades de *fairness*. Um exemplo deste tipo de propriedade que pode ser expressada em LTL, mas não em CTL, é a propriedade descrita pela fórmula A(GFp $\rightarrow$ GFp). A prova da ausência de equivalência total entre essas duas linguagens está fora do contexto do trabalho aqui apresentado e pode ser vista em [Mai00].

O poder de expressividade destas duas linguagens não é uma questão meramente teórica, mas também uma questão de usabilidade. Ferramentas de verificação de modelos são usadas por engenheiros dentro de grandes empresas, como Intel, IBM, entre outras, para garantir que sistemas complexos tais como protocolos e *designs* de chip funcionam como pretendido. Assim, cada uma destas linguagens é usada em situações diferentes, dependendo do tipo de propriedade que se quer verificar.

Conclusões

Neste apêndice foi apresentada a técnica de verificação de modelos, que serve como embasamento teórico para o trabalho aqui apresentado. Foi mostrada de forma breve como se dão as etapas desta técnica.

Além disso, duas linguagens de lógica temporal, voltadas a descrição de propriedades, foram mostradas, LTL e CTL. Uma rápida descrição sobre elas foi feita, bem como uma comparação entre o poder de expressividade de ambas.

Neste trabalho, optou-se por utilizar a linguagem LTL. Isso ocorreu devido à escolha pelo verificador de modelos utilizado. O verificador de modelos que mais se adequou às necessidades do trabalho aqui apresentado faz uso de LTL para especificação das propriedades. Tal ferramenta é apresentada no Apêndice B.

Apêndice B

O Verificador *Bogor* e a Linguagem *BIR*

Neste apêndice é apresentada uma breve descrição do verificador de modelos escolhido neste trabalho, o verificador Bogor. Também é apresentada a parte da linguagem BIR usada neste trabalho, e algumas partes que podem ser usadas em trabalhos futuros. Para cada estrutura da linguagem é mostrada a respectiva sintaxe e uma explanação sobre a semântica.

Bogor

O Bogor é um arcabouço de software extensível para verificação de modelos. Ele possui os mais novos algoritmos de verificação de modelos usados atualmente e possui um *design* para suportar propósitos gerais e de domínio específico do software, em se tratando de verificação de modelos. Embora existam muitos verificadores de modelo disponíveis, este possui capacidades modernas que o tornam especialmente bem preparado para a verificação de propriedades de uma variedade de artefatos de software também modernos, desde a construção de uma *engine* voltada a um domínio específico, até o seu uso no ensino dos conceitos de verificação de modelos.

Além disso, a ferramenta Bogor possui algumas características que a fizeram o verificador usado neste trabalho:

- suporte a características encontradas em linguagens orientadas a objetos concorrentes, tais como criação dinâmica de *threads* e objetos, herança de objetos e exceções;
- linguagem de modelagem que pode ser estendida com novos tipos primitivos, expres-

sões, e comandos associados a um domínio particular (e.g, sistemas multi-agentes, protocolos de segurança, etc.) e a um nível particular de abstração (e.g, modelos de *design*, código fonte, etc.);

- arquitetura aberta e organizada em módulos que permite a troca, de forma fácil, dos algoritmos de verificação do Bogor por novos algoritmos que podem ser implementados para um domínio específico;

- e *plugin* implementado para o Eclipse com uma variedade de visualizações [1] que, em problemas com necessidades mais específicas, pode ser usado em conjunto com o *plugin* apresentado neste trabalho;

Em resumo, o Bogor não é apenas uma ferramenta de verificação de modelos capaz de manipular construtores de linguagens encontradas em implementações e *designs* de software modernos de larga escala. Ele é um arcabouço de verificação de modelos que possibilita pesquisadores e engenheiros criarem suas próprias *engines* de verificação voltadas para seus domínios específicos.

Com posse destas características, qualquer mudança no foco do trabalho aqui apresentado causará um impacto bem menor do que se fosse utilizado qualquer outro verificador de modelos, grátis e com código aberto, existente.

BIR

Todo modelo possui o seguinte formato: *system <identificador_da_modelagem> { ... }*. A modelagem fica contida entre as chaves. Os identificadores da linguagem BIR estão divididos em: *identificadores normais* e *identificadores de variáveis de tipo*. Há dois tipos de identificadores normais: *identificadores básicos* e *identificadores Bogor*. Identificadores básicos são similares aos identificadores Java. Isto é, eles não podem ter seqüências de caracteres que são iguais as palavras chaves, literais booleanos, literal *null*, e os literais especiais, como *float* e *double* por exemplo.

[1] Como o componente de tradução de contra-exemplos em BIR para contra-exemplos em JaCA não foi implementado ainda, uma alternativa é usar o visualizador de contra-exemplos do próprio Bogor.

Há, geralmente, alguns problemas com identificadores na tradução a partir de uma linguagem de programação, ou a partir da linguagem de anotação como é o caso do trabalho aqui apresentado. Por exemplo, caso um programa Java tivesse uma variável $system$, haveria conflito com a palavra reservada $system$ da linguagem BIR. Para resolver isso, é adotada uma convenção para codificar as variáveis locais Java usando delimitadores "[|"e "|]". Deste modo, é obtido $[|system|]$, em vez de $system$. Exemplos de identificadores válidos em BIR: endereco_residencial, {|endereco residencial|}, (|endereco residencial|), <|endereco residencial|>, [|endereco residencial|], +|endereco residencial|+ e .|endereco residencial|..

Há dois tipos em BIR: *tipos primitivos* e *não primitivos* (para referências). Além dos tipos básicos, BIR oferece tipos genéricos (usado para extensões). BIR também oferece tipos funcionais para suportar sua sub-linguagem funcional de programação.

Tipos primitivos

- Booleanos (*boolean*): podem assumir *true* ou *false*, sendo este último o valor padrão. Os operadores usados para este tipo são: "&&", "||", "==", "!=" e "!" (unário).

- Inteiros (*int* e *long*): variam entre -2147483648 e 2147483647, inclusive. Possui um tipo que "varia": *int* (x,y), que significa que varia do valor 'x' ao valor 'y', inclusive. Caso seja necessário usar um valor fora da faixa, deve-se usar a palavra chave *wrap*. Há uma forma de usar inteiros maiores que a faixa estabelecida: *long*, que varia entre -9223372036854775808 a 9223372036854775807, inclusive. *long* (x,y) funciona de forma análoga a *int* (x,y). O valor padrão para ambos *int* e *long* é 0. Os operadores usados para estes tipos são: "+", "-", "*", "/", "%", "==", "!=", "<", ">=", ">" e ">=". Exemplos: int (0, 8), int wrap (-1, 8), long (0, 10000000000L), long wrap (0, 10000000000L).

- Reais (real e double): possuem como valor padrão 0.0. Os operadores utilizados são: "+", "-", "*", "/", "%", "==", "!=', "<", "<", ">", e ">" (operadores binários).

- Enumerável: o tipo enumerável é definido por símbolos inseridos pelo usuário. Os operadores binários são: "==" e "!=".

Tipos não primitivos

Tipos básicos não primitivos são tipos de referências. Há cinco tipos de não primitivos: *record*, *array*, *string*, *lock*, e extensões de não primitivos. Os valores padrões são sempre *null* e os operadores aplicáveis são "==" e "!=".

- *Record*: usado para modelar classes, exceto métodos. Exemplos: *top throwable record A { int x; }*; *throwable record B { int y; }*; e *record C extends A, B { int z; }*.
- *Array*: representa uma coleção de valores. Os índices variam entre 0 e (*tamanho_do_array* - 1)
- *String*: representa seqüências de caracteres.
- *Lock*: tipo usado para propósitos de sincronização. Usado em abstrações maiores (modela monitores Java). Este tipo contém algumas informações: o descritor da *thread*, informações sobre dono do *lock*; quantas vezes o dono adquiriu o *lock*; um conjunto de espera que contém descritores de *threads* correspondentes a *threads* que estão esperando pelo *lock*; e conjunto de notificação que contém descritores de *threads* que foram notificadas depois de esperar pelo *lock*.

Outros tipos

- Alias: É usado para dar um alias a um tipo declarado. Exemplos: *typealias byte int wrap(0, 255);*; *typealias abc Triple.type<A, B, C>;*; e *typealias abcset Set.type<abc>;*.
- Genérico: tipo que serve para tipos de extensões e argumentos de tipos de extensão de ações e expressões.
- Fun: tipo que suporta expressões funcionais, tais como a linguagem SML [AM91].

Literias

- literais booleanos: *true*, *false*
- literais *char*: 'a', 'b', 'c'

- literais *int*: 1, 2, 3
- literais *long*: 1l, 2l, 3L
- literais *float*: 1.0f, 2f, 3.0F, 4F
- literais *double*: 1.0d, 2d, 3.0D, 4D
- literais *string*: “a”, “b”, “c”

Declarações

O tipo da constante é deduzido a partir do conteúdo declarado. Na Figura B.1, pode ser vista a declaração de algumas constantes. O tipo *int wrap (0,255)* é usado para representar o tipo *byte*, existente na linguagem Java.

```
const Constantes {
  N = 5;                          // int
  M = (int wrap (0, 255)) 5;      // int wrap (0, 255)
  L1 = 5L;                        // long.
  F1 = 1.0F;                      // float.
  D1 = 1.0;                       // double.
  C = 'c';                        // char
  B1 = true;                      // boolean.
  S1 = "str";                     // string.
}
```

Figura B.1: Declaração de constantes em BIR.

O tipo enumerável pode ser declarado como ilustrado na Figura B.2, onde estão declarados os enumeráveis *Dia* e *Cor*.

```
enum Dia {
  Segunda, Terca, Quarta, Quinta, Sexta
}

enum Cor {
  Verde, Azul, Amarelo, Vermelho
}
```

Figura B.2: Declaração de enumeráveis em BIR.

Como mencionado anteriormente, o tipo *record* é usado para modelar apenas os atributos das classes. Na Figura B.3 pode ser visto a declaração de três *records*. A primeira é

um *record* chamado *SuperClasse*. O segundo é chamado *SubClasse* e estende o primeiro *record*. O terceiro e último representa uma exceção em virtude da presença da palavra-chave *throwable*.

```
record SuperClasse { int x; }

record SubClasse extends Superclasse { int y; }

throwable record Excecao {}
```

Figura B.3: Declaração de *record* em BIR.

No caso da declaração de variáveis globais, basta declarar tais variáveis fora do escopo de *records*, funções ou *threads*. Desta forma, o verificador interpretará-las como sendo globais no modelo especificado.

Threads e funções são similares, exceto pelo fato de que funções podem retornar valores. Se o modificador *active* for usado, então a *thread* entrará em execução no estado inicial do sistema, caso contrário, a *thread* só entrará em execução quando for invocada. Multiplicidades podem ser usadas para indicar a quantidade de instâncias da *thread* no estado inicial.

Há dois tipos de corpos de *threads* e de funções em BIR: corpo de baixo nível (desestruturado - similar aos *bytecodes* Java) e corpo de alto nível (estruturado - similar ao código fonte Java). O Bogor trabalha com os corpos de baixo nível. Entretanto, o usuário pode modelar usando a representação de alto nível, pois há uma tradução automática a partir desta representação para a de baixo nível. Como ambos os corpos apresentam os mesmos recursos e o corpo de alto nível é mapeado mais facilmente para a linguagem JaCA, foi escolhido (e será mostrado) apenas este corpo. A seguir são mostradas alguns tipos de sentenças, de expressões e de ações usadas em funções e *threads*.

Sentenças

Na Figura B.4 pode ser visto um exemplo simples da sentença *while*. As instruções que estiverem entre do e end são executadas repetidamente até que a condição, *i < 10* neste caso, seja avaliada em falso.

Na Figura B.5 pode ser visto um exemplo simples da sentença *if*. Sua semântica, assim como a sentença *while*, é idêntica a semântica do *if* em Java. Este exemplo possui

```
system ExemploWhile {
  active thread MAIN() {
    int i := 0;
    while i < 10 do
      ...
      i := i + 1;
    end
  }
}
```

Figura B.4: Exemplo de sentença *while*.

uma pequena diferença dos demais. Há "[3]" logo após a palavra reservada *active*. Como mencionado anteriormente, multiplicidades podem ser usadas para indicar mais de uma instância da *thread* no estado inicial. Neste caso, três instâncias desta *thread* são executadas, compartilhando a variável i.

```
system ExemploIf {
  int i := 0;
  active[3] thread MAIN() {
    if i < 1 do
      i := i + 1;
    else if i < 2 do
      i := i + 2;
    else do
      i := i + 3;
    end
  }
}
```

Figura B.5: Exemplo de sentença *if*.

Na Figura B.6 é ilustrado um exemplo da sentença *choose*. Como pode ser visto, cada uma das condições é indicada com *when*. O verificador de modelos Bogor avalia cada uma da condições. Caso uma delas seja avaliada como verdadeira, as instruções que corresponderem a ela são executadas. Caso mais de uma condição seja avaliada como verdadeira, como é o caso do exemplo ilustrado nesta figura (quando *i=0*), o verificador cria uma ramificação para cada uma das execuções correspondentes as condições avaliadas como verdadeiro. E se nenhuma das condições for avaliada como verdadeira, as instruções do *else* são então executas.

Na Figura B.7 é apresentado o uso da sentença *try*. São mostradas as declarações de dois

```
system ExemploChoose {
  int i := 0;
  active thread MAIN() {
    choose
      when <i < 1> do
        i := i + 1;
      when <i < 2> do
        i := i + 2;
      else do
        i := i + 3;
    end
  }
}
```

Figura B.6: Exemplo de sentença *choose*.

registros, um simples (registro *A*) e um representando uma exceção (registro *NPE*). Neste caso, um objeto do tipo *A* é criado mas não é instanciado (*a := new A;*). Ao somar 10 à variável *x*, é levantada uma exceção do tipo *NullPointerException*, assim como em Java. Esta exceção é guardada na variável *npe*, do tipo *NPE*, declarada no início da *thread*.

```
system ExemploTry {
  throwable record NPE { }

  record A { int x; }

  active thread MAIN() {
    A a;
    NPE npe;
    try
      a.x := a.x + 10;
    catch (NPE npe)
      a := null;
    end
  }
}
```

Figura B.7: Exemplo de sentença *try*.

Na Figura B.8 é mostrada a sentença *return*. Similar a Java, esta sentença é usada para sair de funções e *threads*. Se a função tiver um tipo de retorno, então a sentença *return* deve ter uma expressão cujo tipo resultante seja compatível com o tipo de retorno.

Na Figura B.9 é mostrada a sentença *skip*. O *skip* possui a mesma funcionalidade que o *break* possui na linguagem Java, ele move a execução para o próximo ponto de controle.

```
system ExemploReturn {
  active thread MAIN() {
    int result;
    result := fact(3);
  }

  function fact(int x) returns int {
    return x <= 1 ? 1 : x * fact(x - 1);
  }
}
```

Figura B.8: Exemplo de sentença *return*.

```
system SkipExample {
  active thread MAIN() {
    int i;
    choose
      do
        i := 8;
      do
        skip;
    end
  }
}
```

Figura B.9: Exemplo de sentença *skip*.

Expressões

Abaixo são mostradas duas expressões suportadas por BIR, uma de criação de *record* e outra de criação de *array*:

- *Registro reg := new Registro;* - cria e instancia um objeto do tipo *Registro*.
- *int[] numeros := new int[6];* - cria um array de números inteiros com capacidade para seis elementos.

Em expressões de acesso a campos, caso o *array* ou registro seja nulo, é levantada uma exceção de ponteiro nulo, semelhante a exceção *NullPointerException* em Java. Exemplos de acesso:

- Acesso ao campo *campo* do registro *reg* - *reg.campo*.
- Acesso ao primeiro elemento de um *array* - *numeros[0]*.

- Acesso ao primeiro elemento da segunda coluna de uma matriz - *matriz[0][1]*.

Também é possível fazer *cast* de tipos em BIR, assim como em algumas linguagens de programação. Exemplos: *(int) wrap (0, 255) 1*, *(float) 4*, e *(Registro2) registro1*. Outro tipo de expressão existente são as expressões de teste de *lock*. Estas expressões são usadas para obter informações sobre o estado de um certo *lock*. Exemplos:

- *lockAvailable(l)* - testa se um *lock* está livre.
- *hasLock(l)* - testa se uma *thread* em execução detém (possui) um *lock*.
- *wasNotified(l)* - testa de houve uma notificação para um *lock*.

Além das expressões de teste de *lock*, existe uma expressão de teste de *thread*. Tal expressão é usada para determinar o estado de uma *thread*: *wasTerminated(myTID)*.

Ações

Atribuição de valores para variáveis, campos de *record*, *arrays*, ou elementos de array são feitas por meio de *ações de atribuição*. Exemplos: *var := 1;*, *reg.campo := 1;*, *array1 := array2;*, e *array1[0] := 5;*.

A ação de *assert* é usada para verificar se um *assert* é avaliado como verdadeiro. Caso não seja avaliado como verdadeiro, uma seqüência de execuções é retornada para o usuário. Exemplos: *assert f(x);*, *assert x > 0;*, e *assert ((x > 0) || (y = 7));*.

Outra ação existente é a ação de *assume*. Tal ação é usada para filtrar caminhos, isto é, se a condição de *assume* é avaliada como falsa, o Bogor irá efetuar *backtracking* para explorar outro caminho do espaço de estados, otimizando a verificação. Exemplos: *assume f(x);*, *assume x > 0;*, e *assume ((x > 0) || (y = 7));*.

Existem ações usadas para operações especiais, como a ação de operação de *lock*. Esta operação é usada para mudar o estado de um valor de *lock*. As ações usadas neste trabalho são: *lock*, *unlock*, *wait*, *notify*, *notifyall*. Todos eles possuem a mesma semântica de seus homônimos da linguagem Java, exceto *lock* e *unlock* que são usados para obter e liberar o lock de um determinado objeto. Exemplo: *lock(objeto);*.

Há também ações para levantar exceções, ação de *throw exception*. Usada para levantar um valor de *record "throwable"*, semelhante em Java ao levantamento de exceções. Exemplos: *throw r*; e *throw new ThrowableRecord;*.

Há outras duas ações relevantes. Uma delas é a ação de início de *thread*, usada para criar uma instância de uma *thread* e retornar o valor do descritor desta *thread*. Exemplos: *start T(5);* e *myTID := start T();*. Outra é a ação de saída de *thread*. Usada para finalizar a execução da *thread* atual: *exit;*.

Apêndice C

Equivalências entre *JaCA* e *BIR*

Neste apêndice são mostrados os mapeamentos entre as estruturas no formato da linguagem JaCA e as estruturas no formato da linguagem BIR. A maioria delas é mapeada de forma direta, diferenciando-se apenas em questão de sintaxe. Eventualmente uma breve descrição sobre alguns dos mapeamentos é apresentada.

Anotações ao nível de classe

Na Figura C.1, é apresentada uma anotação ao nível de classe e sua representação na linguagem BIR, em forma de *record*. Como mencionado no Apêndice B, pode haver problemas com palavras reservadas de BIR ao se traduzir a partir de uma outra linguagem. Desta forma, foi convencionado, neste trabalho, que os atributos de *records* fossem delimitadas com "[|"e "|]".

```
@att(
  int id;
  String nome;
  boolean aprovado;
)
public class Estudante { ... }
```

(a) Código JaCA.

```
record (|Estudante|) {
  int [|id|];
  string [|nome|];
  boolean [|aprovado|];
}
```

(b) Código BIR.

Figura C.1: Equivalência em anotação ao nível de classe.

Como na linguagem BIR não é possível que o campo de um *record* possua valor inicial, uma variável global é criada no modelo além do campo no *record*. Na Figura C.2(a) pode

ser visto um exemplo de atributo com valor inicial determinado. Sempre que um objeto for instanciado, é atribuído ao campo o valor inicial guardado na variável global.

```
@att(
  int id;
  String nome;
  boolean aprovado = false;
)
public class Estudante { ... }
```

(a) Código JaCA.

```
boolean .|Estudante.aprovado|. := false;

record (|Estudante|) {
  int [|id|];
  String [|nome|];
  boolean [|aprovado|];
}
```

(b) Código BIR.

Figura C.2: Equivalência em anotação ao nível de classe com atributo não nulo.

Na Figura C.2(b), a variável booleana chamada *aprovado*, pertencente à classe *Estudante*, foi traduzida para uma variável global no modelo, chamada .|*Estudante.aprovado*|.. Sempre um *record* for instanciado no modelo, logo em seguida a variável global correspondente ao campo não nulo é atribuída ao campo. Ou seja, no modelo, logo após a sentença *estudante := new (|Estudante|);*, é executada a atribuição: *estudante.[|aprovado|] :=* .|*Estudante.aprovado*|.;.

Variáveis estáticas, ou de classe, também são transformadas em variáveis globais no modelo. Contudo, não são traduzidas para o *record*. Desta forma, sempre que alguma referência a alguma variável estática for feita nas anotações, os algoritmos contidos nas classes Java que representam as entidades BIR alteram esta referência pela própria variável global do modelo.

Anotações ao nível de método

Sempre que um método *main* for anotado com a *tag @start*, todo o seu comportamento descrito dentro de *@behavior* será traduzido para a linguagem BIR em forma de *thread*. São adicionados as palavras-chave *"active main thread"*. A palavra *thread* indica que será uma linha de execução do modelo; a palavra *main* indica que será a linha principal; e *active* indica que a execução desta função será iniciada juntamente com o inicio da execução do modelo. Todo modelo comportamental deve possuir pelo menos uma função *active thread*.

Se por acaso o método *main* de uma classe chamada *Classe* for anotada com *@start*, como em *@start public static void main(String[] args) { ... }*, a função correspondente em

BIR terá a forma: *active main thread {|Classe.main()|}() { ... }*.

Anotações comportamentais

Na FiguraC.3 pode ser visto a equivalência entre os códigos JaCA e BIR para criação de objetos. Em BIR, todas as variáveis de uma função devem ser obrigatoriamente declaradas no início da mesma. Desta forma, caso uma variável seja criada em qualquer parte de uma anotação, ela será declarada no início da função no modelo após a tradução. Na figura em questão, um objeto *estudante* é criado e instanciado em alguma parte da anotação. Ao ser traduzida para o modelo, esta sentença é dividida em duas etapas. Primeiramente, a variável *[|estudante|]* é declarada no início da função. Depois, no ponto de execução do modelo que corresponde ao ponto de instanciação da anotação, é atribuída à variável *[|estudante|]* o valor retornado pela função *{|Estudante()|}()*, que corresponde ao construtor da classe *Estudante*.

```
@behavior(
  ...
  Estudante estudante = new Estudante(5);
  ...
)
public static void main(String[] args) {
  ...
  Estudante estudante = new Estudante(5);
  ...
}
```

(a) Código JaCA.

```
active main thread {|Classe.main()|}() {
  (|Estudante|) [|estudante|];
  .|Classe.main()|. := true;
  ...
  [|estudante|] := {|Estudante(int)|}(5);
  ...
  .|Classe.main()|. := false;
}
```

(b) Código BIR.

Figura C.3: Equivalência em anotação ao nível de método - criação de objetos.

Como pode ser visto, há atribuições a duas variáveis no código BIR que não há no código JaCA. Estas variáveis são inseridas como informação adicional em toda e qualquer função

(ou *thread*) que possua proposição referente a sua invocação, execução e/ou finalização em alguma anotação de especificação de propriedade. São elas que possibilitam a verificação de ordem de execução e de invocação de métodos. Estas variáveis possuem o formato: .|*NomeDaClasse.NomeDoMetodo(assinatura)*|..

Na FiguraC.4 pode ser visto a tradução de invocação de métodos. Em BIR, não há ligação entre *record* e função como há em Java entre classe e método. Assim, na tradução de métodos não estáticos, é sempre adicionado à função um parâmetro do tipo da classe ao qual o método pertence. É neste parâmetro é que são executadas as computações da função.

```
@behavior(
  ...
  estudante.getName();
  ...
)
public static void main(String[] args) { ... }
```

(a) Código JaCA.

```
active main thread {|Classe.main()|}() {
  ...
  {|Estudante.getName()|}([|estudante|]);
  ...
}
```

(b) Código BIR.

Figura C.4: Equivalência em anotação ao nível de método - invocação de método.

A Figura C.5 ilustra a tradução de um método não estático. Vale a penas mencionar mais uma vez que a função em BIR possui um parâmetro a mais: o objeto no qual serão executadas as computações.

Há várias maneiras de se modelar um laço. Na Figura C.6 é ilustrada uma delas. Como mencionado também naquele capítulo, não é necessário definir uma variável *i* para formar a condição de parada do laço. Qualquer expressão que resulte num inteiro pode ser usada. A variável *i*, ou *j* se for o caso de já existir alguma *i* na função, é automaticamente inserida no modelo BIR.

É possível também estabelecer várias condições para o laço, separando-as com o símbolo '|'. Na Figura C.7 é apresentada a tradução da sentença *loop* com múltiplas condições de parada.

```
@behavior( return this.name; )
public String getName(){
  return this.name;
}
```

(a) Código JaCA.

```
function {|Estudante.getName()|}((|Estudante|) this) returns String {
  .|Estudante.getName()|. := true;
  .|Estudante.getName()|. := false;
  return this.[|name|];
}
```

(b) Código BIR.

Figura C.5: Equivalência em anotação ao nível de método - métodos não estáticos.

```
@behavior(
  ...
  int i = 0;
  loop(i<estudante.notas.length){
    ...
    i++;
  }
  ...
)
public float somarNotas(Estudante estudante){ ... }
```

(a) Código JaCA.

```
function {|Estudante.somarNotas()|}((|Estudante|) this)
                                          returns float {
  int i;
  ...
  i := 0;
  while (i < this.notas.length) do
    ...
    i := i+1;
  end
  ...
}
```

(b) Código BIR.

Figura C.6: Equivalência em anotação ao nível de método - laço.

Graças a um recurso da linguagem BIR e de seu verificador de modelos Bogor, múltiplas ramificações são criadas no espaço de estados criado pela execução do modelo, uma

```
@behavior(
  ...
  loop(4|true){
    ...
  }
  ...
)
public float somarNotas(Estudante estudante){ ... }
```

(a) Código JaCA.

```
function {|Estudante.somarNotas()|}((|Estudante|) this)
                                          returns float {
  int i;
  .|Estudante.somarNotas()|. := true;
  ...
  choose
    when <true> do
      i := 0;
      while (i < 4) do
        ...
        i := i+1;
      end
    when <true> do
      while (true) do
        ...
      end
    else do;
  end
  ...
  .|Estudante.somarNotas()|. := false;
  return ...;
}
```

(b) Código BIR.

Figura C.7: Equivalência em anotação ao nível de método - laço com ramificações.

ramificação para cada condição. Isso é feito devido a estrutura *choose*. Como explanado no Apêndice B, uma ramificação é criada para cada condição avaliada como verdadeira. No caso do *loop* com várias condições, um *when* com condição igual a *true* é criado para cada condição modelada no *loop*.

Na Figura C.8 é mostrado a tradução de uma sentença *choice*, que modela desvios condicionais.

Para iniciar a execução de alguma *thread* (linha de execução), basta usar a sentença *start{*

```
@behavior(
  ...
  choice{
    (i > 0): { ... }
    else: {
      choice{
        (i < 0): { ... }
        else;
      }
    }
  }
  ...
)
```

(a) Código JaCA.

```
...
choose
  when <i > 0> do
    ...
  else do
    choose
      when <i < 0> do
        ...
      else do;
    end
end
...
```

(b) Código BIR.

Figura C.8: Equivalência em anotação ao nível de método - desvio condicional.

... *}*. Entre as chaves deve estar o objeto correspondente a *thread* seguido de ponto e vírgula (;). O formato na lingagem BIR de *start{ threadA; threadB; }* é: *start {|NomeDaClasseDaThreadA.run()|}(threadA); start {|NomeDaClasseDaThreadB.run()|}(threadA);*.

E em relação a cláusula *abstract* utilizada para redefinir assinaturas de métodos e de construtores, e retorno de métodos, é a mesma tradução mostrada na Figura C.5. A diferença é que com a presença de *abstract*, os algoritmos extraem informações somente da anotação e desconsideram completamente qualquer informação que possa ser extraída do método.

Anotações de especificações de propriedades

Na Tabela C.1, apresentada a seguir, mostra o mapeamento entre os símbolos usados nas anotações e as funções BIR equivalentes.

Na Figura C.9 é mostrado um exemplo de tradução de uma anotação de especificação de propriedade em uma função BIR. Note que em JaCA, as propriedades são separadas por ponto e vírgula (;). Estas mesmas propriedades são transformadas numa única propriedade formada pela conjunção das propriedades anotas em JaCA.

Na Figura C.10, é ilustrado como funciona a verificação de ordem de invocação/chamada de métodos. As cláusulas definidas na linguagem JaCA, *exec* e *invoc*, determinam como será a fórmula LTL final (no formato de função BIR).

Em LTL padrão, a fórmula da propriedade presente na Figura C.10 ficaria no seguinte

Operador LTL	Função BIR
[]	always
<>	eventually
!	negation
U	until
R	release
->	implication
<->	equivalence
&&	conjunction
||	disjunction

Tabela C.1: Mapeamento entre operadores LTL e funções BIR.

formato:

```
[](
  (
    ([](mAExecutadoInicio -> <>mAExecutadoFim))
    &&
    !mBInvocadoFim
  )
  ->
  <>mBInvocadoInicio
)
```

O artifício utilizado neste trabalho para verificar a ordem de execução dos métodos funciona apenas quando uma única linha de execução que invoca os métodos verificados está *rodando*. Supondo que é desejado verificar uma certa propriedade sobre o comportamento do sistema em relação a linha de execução *t1*. Tal propriedade exige que o método *mB()* seja invocado apenas depois que o método *mA()* seja invocado, mas exige também que *mB()* seja invocado antes de *mA()* ser finalizado. Com base nestas informações, o artifício falha no seguinte caso:

1. Linha de execução *t1* inicia método *mA()*: variável .|Classe.mA()|. se torna *true*.
2. Linha de execução *t2* inicia método *mA()*: variável .|Classe.mA()|. já é *true*, mas isso não interfere na verificação.

3. Linha de execução *t2* finaliza método *mA()*: variável .|Classe.mA()|. se torna *false*.
4. Linha de execução *t2* inicia método *mB()*: variável .|Classe.mA()|. continua *false*.
5. Linha de execução *t1* inicia método *mB()*: variável .|Classe.mA()|. já é *false* quando ainda deveria ser *true*.

Assim, a propriedade especificada é avaliada como falsa. O modelo não a satisfaz. Isso se dá devido a uma característica da linguagem BIR. Ela não é capaz de associar linhas de execução a funções.

Deste modo, a verificação de ordem de métodos deve ser usada em situações específicas. Caso seja necessário fazer este tipo de verificação, o desenvolvedor deve modelar (anotar) seu código com um cenário bem definido.

```
@property(
  propdec{
    mediaSuficiente :: estudante.media>=7;
    estudanteAprovado :: estudante.aprovado;
  }
  spec{
    [](mediaSuficiente -> <>estudanteAprovado);
    <>(estudanteAprovado || !estudanteAprovado);
  }
)
public static void main(String[] args){ ... }
```

(a) Código JaCA.

```
fun PropriedadeClasseMain() returns boolean =
  LTL.temporalProperty(
    Property.createObservableDictionary(
      Property.createObservableKey(
        "mediaSuficiente", [|estudante|].[|media|] >= 7
      ),
      Property.createObservableKey(
        "estudanteAprovado", [|estudante|].[|aprovado|]
      )
    ),
    LTL.conjunction(
      LTL.always(
        LTL.implication(
          LTL.prop("mediaSuficiente"),
          LTL.eventually(LTL.prop("estudanteAprovado"))
        )
      ),
      LTL.eventually(
        LTL.disjunction(
          LTL.prop("estudanteAprovado"),
          LTL.negation(LTL.prop("estudanteAprovado"))
        )
      )
    )
  );
```

(b) Código BIR.

Figura C.9: Equivalência em anotação a nivel de método - especificação de propriedade.

```
@property(
  propdec{
    mAExecutado :: exec(Classe.mA());
    mBInvocado :: invoc(Classe.mB());
  }
  spec{
    [](mAExecutado -> <>mBInvocado);
  }
)
```

(a) Código JaCA.

```
fun PropriedadeClasseMain() returns boolean =
  LTL.temporalProperty(
    Property.createObservableDictionary(
      Property.createObservableKey(
        "mAExecutadoInicio", .|Classe.mA()|. == true
      ),
      Property.createObservableKey(
        "mAExecutadoFim", .|Classe.mA()|. == false
      ),
      Property.createObservableKey(
        "mBInvocadoInicio", .|Classe.mB()|. == true
      ),
      Property.createObservableKey(
        "mBInvocadoFim", .|Classe.mB()|. == false
      )
    ),
    LTL.always(
      LTL.implication(
        LTL.implication(
          LTL.conjunction(
            LTL.always(
              LTL.prop("mAExecutadoInicio"),
              LTL.eventually(LTL.prop("mAExecutadoFim"))
            ),
            LTL.prop("mBInvocadoFim")
          )
        ),
        LTL.eventually(LTL.prop("mBInvocadoInicio"))
      )
    )
  );
```

(b) Código BIR.

Figura C.10: Equivalência - especificação de propriedade (ordem de invocação/chamada de métodos).

Apêndice D

Experiência com o Desenvolvedor

Neste apêndice é mostrado um pequeno relatório sobre a experiência de contato do desenvolvedor, leigo em métodos formais, com a linguagem de anotação JaCA, definida neste trabalho. Primeiramente é apresentada a metodologia adotada nesta experiência. Em seguida são mostrados os resultados obtidos a partir da fase de anotação do código feita pelo desenvolvedor. E por último, é mostrado o questionário de avaliação que foi passado ao desenvolvedor.

Metodologia

Num primeiro momento, a sintaxe da linguagem de anotação e alguns exemplos de modelagem foram apresentados ao desenvolvedor por meio de *slides*, em uma aula de aproximadamente quarenta minutos. O Capítulo 3 deste trabalho foi passado a ele como material de consulta sobre a sintaxe e semântica das estruturas da linguagem de anotação. Além disso, algumas poucas intervenções foram necessárias para esclarecer algumas dúvidas sobre o processo de modelagem.

Experiência

O desenvolvedor teve uma pequena dificuldade inicial em saber o que modelar exatamente em seu sistema. Após uma breve iteração, foi coletado do usuário o que ele queria verificar. O próprio usuário definiu quais métodos e classes deveriam ser anotadas, baseado no que se queria verificar.

Apesar de constar no material de consulta passado ao usuário, e de estar presente nos *slides* da aula apresentada (mas não ter sido explicitamente citado), o usuário não fez uso dos ponto-e-vírgulas (;) ao final de cada uma das sentenças. Este erro nas anotações foi rápida e facilmente eliminado. O

próprio *plugin* forneceu a informação do que estava errado as anotações.

Foi detectada uma característica interessante nesta experiência. A necessidade em verificar dois projetos que interagem entre si. A ferramenta aqui desenvolvida (o *plugin*) gera dois modelos separados, um para cada projeto selecionado. Numa próxima versão da ferramenta, ao selecionar dois projetos, um único modelo comportamental será gerado.

Uma última dificuldade do desenvolvedor foi em relação a questão do uso de *API* da qual não se possui o código fonte. Foi necessário explicar ao usuário como contornar esta situação por meio de classes e métodos *stub*. O tempo total de anotação e de aprendizado da linguagem de anotação foi de aproximadamente uma hora e quinze minutos.

Questionário de avaliação

1. Atribua notas de 0 a 10 nos seguintes quesitos:

   (a) Grau de dificuldade em aprender a sintaxe da linguagem: 4.

   (b) Poder de expressividade (modelagem) baseado no estudo realizado: 8.

   (c) Grau de dificuldade em se modelar: 5.

2. Alguma limitação em relação à modelagem realizada?
   Não

3. Caso tenha tido algum problema com limitação, como contornou? Abstraiu?

4. Acredita que o grau de dificuldade em se modelar, usando a linguagem de anotação experimentada, poderia ser reduzido com a prática?
   Definitivamente sim.

5. Quais linguagens de modelagem formal você conhece ou já leu algo sobre?
   Promela, Redes de Petri, Alloy e SMV.

6. Acharia mais fácil ou cômodo modelar usando alguma delas?
   Não, a linguagem de modelagem utilizada é mais próxima do sistema que eu desenvolvi. Talvez Alloy pudesse ter sido usada. Mas não há uma técnica que a integre ao ciclo de desenvolvimento como o este aqui.

7. Algum outro comentário, crítica, ou sugestão, para a melhoria da linguagem?
   Não

Apêndice E

Código e anotações dos sistemas estudados

Neste apêndice são mostradas as anotações e o código fonte das aplicações usadas nos experimentos realizados neste trabalho. Apenas as classes e métodos anotados são apresentados. Assim, todo o código não anotado foi omitido por não fazer parte dos modelos comportamentais analisados.

Servidor de serviços

```
//----------------------------------------------------------------------
package org.percomp.bluetoothserver.common;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.Vector;

import javax.microedition.io.Connector;
import javax.microedition.io.StreamConnection;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
 * @author Loreno Oliveira
 */
@att(StreamConnection conn;)
public class BluetoothConnection {

  private final String TOKEN_DELIMITER = "^";
  private final String MESSAGE_TAIL = "$";
  private final String END_OF_MESSAGE = "#";
  private final int    MESSAGE_SIZE = 16;//bytes

  private StreamConnection conn;
  private DataInputStream dis;
  private DataOutputStream dos;

  @behavior(
    this.conn = conn;
    conn.openDataOutputStream();
    conn.openDataInputStream();)
  public BluetoothConnection( StreamConnection conn ) throws BluetoothConnectionException {
    this.conn = conn;
    try {
```

```
    dos = conn.openDataOutputStream();
    dis = conn.openDataInputStream();
  } catch( IOException e ) { throw new BluetoothConnectionException( e.getMessage() ); }
}

@behavior(
conn.close();
conn = null;)
public void closeConnection() {
  try {
    dos.close();
    dis.close();
    conn.close();
    conn = null;
  } catch( Throwable e ) { e.printStackTrace(); }
}

@behavior(
  this.convertObjectToBytes();
  this.transmitDiscardServiceMessage();)
public void discardClient() throws BluetoothConnectionException, IOException {
  byte[] data = convertObjectToBytes( "bye bye!" + END_OF_MESSAGE );
  transmitDiscardClientMessage( data );
}

@behavior(
  abstract public void transmitDiscardClientMessage(){
    this.transmitDiscardServiceMessage();
  })
private void transmitDiscardClientMessage( byte[] data )
  throws BluetoothConnectionException {
  this.transmitDiscardServiceMessage( data );
}

@behavior(
  abstract private void transmitDiscardServiceMessage(){
    this.transmitAnswer();
  })
private void transmitDiscardServiceMessage( byte[] data )
  throws BluetoothConnectionException {
  try {
    transmitAnswer( data );
  } catch( IOException e ) {
    throw new BluetoothConnectionException( e.getMessage() );
  }
}

@behavior(
  abstract public void waitCommandRequest(){
    this.messageWithoutTail();
  })
public String[] waitCommandRequest() throws BluetoothConnectionException {
  Vector fullMessage = new Vector();
  String messageAsString;
  try {
    while( true ) {
      byte[] message = new byte[MESSAGE_SIZE];
      dis.readFully( message );
      messageAsString = messageWithoutTail( new String( message ) );
      if( messageAsString.endsWith( END_OF_MESSAGE ) ) {
        fullMessage.addElement(
          messageAsString.substring( 0, messageAsString.length() - 1 )
        );
        break;
      }
      fullMessage.addElement( messageAsString );
    }
  String[] asTokens = tokenize( getFullMessage( fullMessage ) );
  return asTokens;
  } catch( IOException e ) {
    throw new BluetoothConnectionException( e.getMessage() );
  }
}

@behavior(
  abstract public void answerCommandRequest(){
    this.convertObjectToBytes();
    this.transmitAnswer();
  })
public void answerCommandRequest( String commandResult )
 throws IOException, BluetoothConnectionException {
  byte[] data = convertObjectToBytes( commandResult + END_OF_MESSAGE );
  transmitAnswer( data );
```

```
}

@behavior(abstract private void messageWithoutTail(){task;})
private String messageWithoutTail( String messageAsString ) {
  String messageWithoutTail = messageAsString;
  while( true ) {
    String lastChar = String.valueOf(
      messageWithoutTail.charAt( messageWithoutTail.length() - 1 )
    );
    if( lastChar.equals( MESSAGE_TAIL ) ) {
      messageWithoutTail =
        messageWithoutTail.substring( 0, messageWithoutTail.length() - 1 );
    } else { break; }
  }
  return messageWithoutTail;
}

@behavior(
  abstract private void transmitAnswer(){
    this.fillMessage();
    this.getMessages();
  })
private void transmitAnswer( byte[] data )
  throws IOException, BluetoothConnectionException {
  byte[] dataOnTheRightLenght = fillMessage( data );
  byte[][] messages = null;
  messages = getMessages( dataOnTheRightLenght );
  for( int i = 0 ; i < messages.length ; i++ ) {
    dos.write( messages[i] );
    dos.flush();
  }
}

@behavior(
  abstract private void transmitDataAndGetAnswer(){
    this.fillMessage();
    this.getMessages();
  })
private String transmitDataAndGetAnswer( byte[] data )
 throws IOException, BluetoothConnectionException {
  byte[] dataOnTheRightLenght = fillMessage( data );
  byte[][] messages = null;
  messages = getMessages( dataOnTheRightLenght );
  for( int i = 0 ; i < messages.length ; i++ ) {
    dos.write( messages[i] );
    dos.flush();
  }
  return waitAnswer();
}

@behavior(abstract private void buildString(){task;})
private String buildString( Vector fullMessage ) {
  String response = "";
  for( int i = 0 ; i < fullMessage.size() ; i++ ) {
    String mes = (String)fullMessage.elementAt( i );
    response += mes;
  }
  return response;
}

@behavior(abstract private void getMessages(){task;})
private byte[][] getMessages( byte[] dataOnTheRightLenght ) {
  int numberOfMessages = dataOnTheRightLenght.length / MESSAGE_SIZE;
  byte[][] messages = new byte[numberOfMessages][MESSAGE_SIZE];
  int pos = 0;
  for( int i = 0 ; i < numberOfMessages ; i++ ) {
    byte[] message = new byte[MESSAGE_SIZE];
    for( int j = 0 ; j < MESSAGE_SIZE ; j++ ) {
      message[j] = dataOnTheRightLenght[pos++];
    }
    messages[i] = message;
  }
  return messages;
}

@behavior(
  abstract private void fillMessage(){
    choice{
      (true):{ return; }
      (true):{
        this.getExtraSize();
        return;
      }
```

```
      }
    })
  private byte[] fillMessage( byte[] data ) {
    if( data.length % MESSAGE_SIZE == 0 ) {//the size is ok
      return data;
    } else {
      int extraSize = getExtraSize( data.length );
      byte[] newData = new byte[ data.length + extraSize ];
      for( int i = 0 ; i < data.length ; i++ ) {
        newData[i] = data[i];
      }
      for( int i = data.length ; i < newData.length ; i++ ) {
        newData[i] = MESSAGE_TAIL.getBytes()[0];//this char has only one byte
      }
      return newData;
    }
  }

  @behavior(abstract private void getExtraSize(){task;})
  private int getExtraSize( int originalSize ) {
    for( int i = 1 ; i < 16 ; i++ ) {
      if( (originalSize + i) % MESSAGE_SIZE == 0 ) {
        return i;
      }
    }
    return -1000;
  }

  @behavior(abstract public void convertObjectsToBytes(){task;})
  private byte[] convertObjectsToBytes( String command, String[] args ) {
    String rawData = command;
    for( int i = 0 ; i < args.length ; i++ ) {
      rawData += TOKEN_DELIMITER + args[i];
    }
    rawData += END_OF_MESSAGE;
    return rawData.getBytes();
  }
}

//-------------------------------------------------------------------------
package org.percomp.bluetoothserver.common;

import javax.bluetooth.UUID;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
* @author Loreno Oliveira
*/
@att(String serviceAlias; UUID serviceUUID;)
public class ServiceEntry {

  private String serviceAlias;
  private UUID serviceUUID;
  private Class serviceConsumerClass;

  @behavior(
    abstract public ServiceEntry(String serviceAlias,UUID serviceUUID){
      this.serviceAlias = serviceAlias;
      this.serviceUUID = serviceUUID;
    })
  public ServiceEntry( String serviceAlias, UUID serviceUUID, Class serviceConsumerClass ) {
    this.serviceAlias = serviceAlias;
    this.serviceUUID = serviceUUID;
    this.serviceConsumerClass = serviceConsumerClass;
  }

  @behavior(return this.serviceAlias;)
  public String getServiceAlias() {
    return this.serviceAlias;
  }

  @behavior(return this.serviceUUID;)
  public UUID getServiceUUID() {
    return serviceUUID;
  }

  @behavior(
    SampleServiceConsumer sc = new SampleServiceConsumer();
    sc.setBluetoothConnection();
    return sc;)
  public ServiceConsumer getServiceConsumer( BluetoothConnection conn ) {
```

```
    ServiceConsumer sc = null;
    try {
      sc = (ServiceConsumer) serviceConsumerClass.newInstance();
    } catch( IllegalAccessException e ) { e.printStackTrace();
    } catch( InstantiationException e ) { e.printStackTrace(); }
    sc.setBluetoothConnection( conn );
    return sc;
  }
}

//-------------------------------------------------------------------------
package org.percomp.bluetoothserver.common;

import org.percomp.bluetoothserver.stubs.UUID;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
* @author Loreno Oliveira
*/
@att(static ServiceEntry[3] services;)
public class ServiceIndex {
  public static ServiceEntry[] services = new ServiceEntry[3];

  @behavior(
    this.services[0] := ServiceEntry(new UUID(),"s1");
    this.services[1] := ServiceEntry(new UUID(),"s2");
    this.services[2] := ServiceEntry(new UUID(),"s3");
    return this.services;)
  public static ServiceEntry[] allServices() {
    services[0] = new ServiceEntry(
      "serviço de teste 1",new UUID("12345678901234567890123456789012", false ),
      SampleServiceConsumer.class );
    services[1] = new ServiceEntry(
      "serviço de teste 2",new UUID("34567890123456789012345678901234", false ),
      SampleServiceConsumer.class );
    services[2] = new ServiceEntry(
      "serviço de teste 3",new UUID("67890123456789012345678901234567", false ),
      SampleServiceConsumer.class );
    return services;
  }
}

//-------------------------------------------------------------------------
package org.percomp.bluetoothserver.main;

import org.percomp.bluetoothserver.common.ServiceEntry;
import org.percomp.bluetoothserver.common.ServiceIndex;
import org.percomp.bluetoothserver.server.ServerThread;

import core.jaca.annotations.behavior;
import core.jaca.annotations.property;
import core.jaca.annotations.start;

/**
* @author Loreno F. de Oliveira, loreno@dsc.ufcg.edu.br
*
*/
public class Main {
  @behavior(serverThread.st_requested = true;)
  public static void invokeService(ServerThread serverThread){
    serverThread.st_requested = true;
  }

  @property(
    propdec{
      InvocaFinalizarServidor :: invoc(ServerThread.abortServer());
      ExecutaFinalizarServidor :: exec(ServerThread.abortServer());
      ServDoClienteAbortado :: serviceDispatchers[0].serviceAborted == true;
      ServDoServidorAbortado :: st1.service == null;
      ServidorAbortado :: st1.serverIsUp == false;
      InvocaAbortarSevico :: invoc(ServiceDispatcher.abortService());
      ExecutaAbortarSevico :: exec(ServiceDispatcher.abortService());
      IniciaTransMsg :: invoc(BluetoothConnection.transmitDiscardClientMessage());
      MsgTransmitida :: exec(BluetoothConnection.transmitDiscardClientMessage());
      MsgFimDeServico :: exec(BluetoothConnection.messageWithoutTail());
      FinalizaAbortarSevico :: fine(ServiceDispatcher.abortService());
      serv1Ok :: st1.serverIsUp;
      serv2Ok :: st2.serverIsUp;
      serv3Ok :: st3.serverIsUp;
      serv1Finished :: !st1.serverIsUp;
      serv2Finished :: !st2.serverIsUp;
```

```
      serv3Finished :: !st3.serverIsUp;
    }
    spec{
      [](InvocaFinalizarServidor ->
        <>(ServDoClienteAbortado ^ <>ServDoServidorAbortado));
      [](ExecutaFinalizarServidor ->
        (ServidorAbortado && ServDoServidorAbortado && ServDoClienteAbortado));
      [](InvocaAbortarSevico -> <>IniciaTransMsg);
      [](!MsgTransmitida -> !ExecutaAbortarSevico);
      [](MsgFimDeServico -> <>FinalizaAbortarSevico);
      <>(ExecutaAbortarSevico);
      [](serv1Ok -> <>serv1Finished);
      [](serv2Ok -> <>serv2Finished);
      [](serv3Ok -> <>serv3Finished);
   }
  }
  @behavior(
    ServiceEntry[] services = ServiceIndex.allServices();
    ServerThread st1 = new ServerThread(services[0]);
    ServerThread st2 = new ServerThread(services[1]);
    ServerThread st3 = new ServerThread(services[2]);
    start{ st1; st2; st3; }
    invokeService(st1);)
  @start
  public static void main( String[] args ) {
    ServiceEntry[] services = ServiceIndex.allServices();
    ServerThread thread = new ServerThread( services[0] );
    for( int i = 1 ; i < services.length ; i++ ) {
      thread = new ServerThread( services[i] );
      thread.start();
    }
    invokeService(thread);
  }
}

//-------------------------------------------------------------------------
package org.percomp.bluetoothserver.server;

import java.io.IOException;
import java.util.Vector;

import javax.bluetooth.DataElement;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.ServiceRecord;
import javax.bluetooth.UUID;
import javax.microedition.io.Connector;
import javax.microedition.io.StreamConnection;
import javax.microedition.io.StreamConnectionNotifier;

import org.percomp.bluetoothserver.common.BluetoothConnection;
import org.percomp.bluetoothserver.common.ServiceEntry;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
* @author Loreno Oliveira
*/
@att(
  boolean serverIsUp = true;
  ServiceEntry serviceEntry;
  StreamConnectionNotifier service;
  static Vector[1] serviceDispatchers;
  public boolean st_requested = false;)
public class ServerThread extends Thread {

  private StreamConnectionNotifier service;
  private boolean serverIsUp = true;
  private ServiceEntry serviceEntry;
  private static Vector serviceDispatchers = new Vector();
  public boolean st_requested = false;

  @behavior(this.serviceEntry = service;)
  public ServerThread( ServiceEntry service ) {
    this.serviceEntry = service;
  }

  @behavior(
    Connector.open();
    LocalDevice device = LocalDevice.getLocalDevice();
    loop(this.serverIsUp){
      this.serviceEntry.getServiceAlias();
```

```
      StreamConnection conn;
      loop(this.st_requested != true){}
      this.st_requested := false;
      conn = service.acceptAndOpen();
      BluetoothConnection btc = new BluetoothConnection( conn );
      ServiceDispatcher dispatcher = new ServiceDispatcher();
      serviceDispatchers[0] = dispatcher;
      dispatcher.serveRequest(btc);
    }
    this.serviceEntry.getServiceAlias();)
  public void run() {
    try {
      service = (StreamConnectionNotifier)Connector.open(
       "btspp://localhost:" + serviceEntry.getServiceUUID().toString() +
       ";name=doesntmatter");
      LocalDevice device = LocalDevice.getLocalDevice();
      device.setDiscoverable( DiscoveryAgent.GIAC );
      while( serverIsUp ) {
        System.out.println("Servico \"" + serviceEntry.getServiceAlias() +
         "\" aguardando clientes...");
        StreamConnection conn = service.acceptAndOpen();
        BluetoothConnection btc = new BluetoothConnection( conn );
        ServiceDispatcher dispatcher = new ServiceDispatcher();
        serviceDispatchers.addElement( dispatcher );
        dispatcher.serveRequest( btc );
      }
    } catch( Exception e ) {
      if(!e.getMessage().equals("Notifier is closed")){e.printStackTrace();}
    }
    System.out.println( "Serviço \"" + serviceEntry.getServiceAlias() + "\" finalizado" );
  }

  @behavior(
    serviceDispatchers[0].abortService();
    this.serverIsUp = false;
    this.service.close();)
  public void abortServer() {
    for( int i = 0 ; i < serviceDispatchers.size() ; i++ ) {
      ServiceDispatcher dispatcher = (ServiceDispatcher)serviceDispatchers.elementAt( i );
      if( dispatcher != null ) { dispatcher.abortService(); }
    }
    try {
      serverIsUp = false;
      service.close();
    } catch( IOException e ) { e.printStackTrace(); }
  }
}


//-------------------------------------------------------------------------
package org.percomp.bluetoothserver.server;

import java.io.IOException;

import org.percomp.bluetoothserver.common.BluetoothConnection;
import org.percomp.bluetoothserver.common.BluetoothConnectionException;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
* @author Loreno Oliveira
*/

@att(
  boolean clientWantsService = true;
  boolean serviceAborted = false;
  BluetoothConnection btc;)
public class ServiceDispatcher extends Thread  {

  private boolean clientWantsService = true;
  private boolean serviceAborted = false;
  private BluetoothConnection btc;

@behavior(task;)
public ServiceDispatcher(){ }

  @behavior(
    this.btc = btc;
    this.clientWantsService = true;
    start{ this; })
  public void serveRequest( BluetoothConnection btc ) {
    this.btc = btc;
    start();
```

```
  }

  @behavior(
    loop(true|this.clientWantsService){
      this.btc.waitCommandRequest();
      this.getArguments();
      choice{
        (!this.clientWantsService): { return; }
        (this.clientWantsService): {
          this.executeCommand();
          this.btc.answerCommandRequest();
        }
      }
      this.clientWantsService = false;
    })
  public void run() {
    try {
      while( clientWantsService ) {
        String[] commandAndArguments = btc.waitCommandRequest();
        String command = commandAndArguments[0];
        String[] arguments = getArguments( commandAndArguments );
        if( command.equals( "bye bye!" ) ) {
          System.out.println( "S: conexao com o cliente fechada" );
          break;
        } else {
          String result = executeCommand( command );
          btc.answerCommandRequest( result );
        }
      }
    } catch( BluetoothConnectionException e ) {
      System.out.println( "Mensagem: \"" + e.getMessage() + "\"" );
      if( e.getMessage().startsWith( "error 10053 during TCP read" ) ) {
        System.out.println( "Servico vai ser descartado devido cancelamento do usuario" );
      } else { e.printStackTrace(); }
    } catch (IOException e) {
      if( e.getMessage().equals( "stream closed" ) ) { }
      else {
        e.printStackTrace();
      }
    }
  }

  @behavior(abstract public void getArguments(){task;})
  private String[] getArguments( String[] commandAndArguments ) {
    if( commandAndArguments.length > 1 ) {
      String[] arguments = new String[commandAndArguments.length - 1];
      for( int i = 1 ; i < commandAndArguments.length ; i++ ) {
        arguments[i - 1] = commandAndArguments[i];
      }
      return arguments;
    } else {
      return new String[]{};
    }
  }

  @behavior(abstract public void executeCommand(){task;})
  private String executeCommand( String command ) {
    System.out.println( "== Executando Serviço ==" );
    try {
      Thread.sleep( 10000 );
    } catch( InterruptedException e ) {
      e.printStackTrace();
    }
    return "12345678";
  }

  @behavior(
    this.serviceAborted = true;
    this.btc.discardClient();
    this.btc.closeConnection();)
  public void abortService() {
    serviceAborted = true;
    try {
      btc.discardClient();
    } catch (BluetoothConnectionException e) {
      e.printStackTrace();
    } catch (IOException e) { e.printStackTrace(); }
      btc.closeConnection();
  }
}

//-----------------------------------------------
package org.percomp.bluetoothserver.stubs;
```

```
import core.jaca.annotations.att;
/**
* @author Elthon Oliveira
*/
@att(int GIAC = 1;)
public class DiscoveryAgent {
  public static int GIAC = 1;
}

//-------------------------------------------------
package org.percomp.bluetoothserver.stubs;

import core.jaca.annotations.att;
/**
* @author Elthon Oliveira
*/
@att()
public class Connection { }

//-------------------------------------------------
package org.percomp.bluetoothserver.stubs;

import java.io.IOException;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
* @author Elthon Oliveira
*/
@att()
public class Connector {
  @behavior(
  abstract public void open(){
    task;
  })
  public static StreamConnection open(String a)
    throws IOException{
    return new StreamConnection();
  }
}


//-------------------------------------------------
package org.percomp.bluetoothserver.stubs;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;
/**
* @author Elthon Oliveira
*/

@att()
public class LocalDevice {

@behavior(
  abstract public void getLocalDevice(){
    task;
  })
  public static LocalDevice getLocalDevice(){
    return new LocalDevice();
  }

  @behavior( task; )
  public void setDiscoverable(int id){ }
}



//-------------------------------------------------
package org.percomp.bluetoothserver.stubs;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;
/**
* @author Elthon Oliveira
```

```
*/

@att()
public class StreamConnection {

  @behavior(
    abstract public void openDataOutputStream(){
      task;
    })
  public DataOutputStream openDataOutputStream()
    throws IOException{ return null; }

  @behavior(
    abstract public void openDataInputStream(){
      task;
    })
  public DataInputStream openDataInputStream()
    throws IOException{ return null; }

  @behavior(
    abstract public static void close(){
      task;
    })
  public static StreamConnection close()
    throws IOException{ return null; }

  @behavior( task; )
  public StreamConnection(){ }
}

//-------------------------------------------------
package org.percomp.bluetoothserver.stubs;

import java.io.IOException;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;
/**
* @author Elthon Oliveira
*/

@att(boolean scn_open = false;)
public class StreamConnectionNotifier extends StreamConnection{

  @behavior(
    this.scn_open = true;
    return this;
  )
  public StreamConnection acceptAndOpen(){
    return new StreamConnection();
  }

  @behavior(
    abstract public void close(){
      task;
    })
    public static StreamConnection close()
      throws IOException{ return new StreamConnection(); }
}

//-------------------------------------------------
package org.percomp.bluetoothserver.stubs;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;
/**
* @author Elthon Oliveira
*/

@att()
public class UUID {

  @behavior( task; )
  public UUID(String a, boolean b){ }
}
```

Monitor de bateria

```
//-------------------------------------------------------------------------
```

```
package org.percomp.wings.applications;

import org.percomp.wings.ContextEvent;
import org.percomp.wings.ContextEventListener;
import org.percomp.wings.ContextInformationRetrievalException;
import org.percomp.wings.MiddlewareFacade;
import org.percomp.wings.UnknownContextKeyException;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

@att(boolean batteryWarnning;)
public class Application1 implements ContextEventListener {

@behavior(
  MiddlewareFacade.getDefault().registerContextListener(false, this);
  Battery use = new BatteryUse();
  start { batteryUse(); }
)
public Application1() {
  try {
    MiddlewareFacade.getDefault().registerContextListener(
    new BatteryLowCondition(), this);
  } catch (UnknownContextKeyException e) { e.printStackTrace();
  } catch (ContextInformationRetrievalException e) { e.printStackTrace(); }
  }
}

//-------------------------------------------------------------------------
package org.percomp.wings.applications;

import org.percomp.wings.CannotAddPluginException;
import org.percomp.wings.MiddlewareFacade;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;
import core.jaca.annotations.property;
import core.jaca.annotations.start;

@att(static int batterylevel = 2;)
public class Applications {
  static int batterylevel = 10;

  @start
  @behavior(
    MiddlewareFacade facade = MiddlewareFacade.getDefault();
    facade.startMiddleware();
    facade.addContextAwarenessPlugin(new DeviceMonitorCAP());
    Application1 app1 = new Application1();
    Application1 app2 = new Application1();)
  public static void main(String[] args) {
    MiddlewareFacade facade = MiddlewareFacade.getDefault();
    facade.startMiddleware();
    try {
      facade.addContextAwarenessPlugin(new DeviceMonitorCAP());
    } catch (CannotAddPluginException e) { e.printStackTrace(); }
    Application1 app1 = new Application1();
    Application1 app2 = new Application1();
    while (true) {}
  }
}

//-------------------------------------------------------------------------
package org.percomp.wings.applications;

import org.percomp.wings.ContextCondition;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

@att()
public class BatteryLowCondition implements ContextCondition {

  @behavior(
    abstract boolean isSatisfied(){
      if (batterylevel < 1) return true;
      else return false;
    })
  public boolean isSatisfied(Object batteryLevel) {
    int level = ((Integer) batteryLevel).intValue();

    if (level <= 5) {
      return true;
```

```
    } else {
      return false;
    }
  }
}

//-------------------------------------------------------------------------
package org.percomp.wings.applications;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

@att()
public class BatteryUse extends Thread{

  @behavior(
    loop (Applications.batterylevel > 0) { Applications.batterylevel--; })
  public void run(){
    while (Applications.batterylevel > 0) { Applications.batterylevel--; }
  }
}

//-------------------------------------------------------------------------
package org.percomp.wings.stub;

import org.percomp.wings.applications.Application1;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

@att()
public class Compor {

  @behavior(
    app.batteryWarnning = boo;
    ContextAwarenessPlugin.registerContextListener(app);)
  public static void publish(boolean boo, Application1 app){}
}

//-------------------------------------------------------------------------
package org.percomp.wings;

import java.lang.reflect.Method;
import java.util.Hashtable;
import java.util.List;
import java.util.Vector;

import net.compor.frameworks.jcf.FunctionalComponent;
import net.compor.frameworks.jcf.service.ProvidedService;
import net.compor.frameworks.jcf.service.ServiceReturn;
import net.compor.frameworks.jcf.service.ServiceSpecification;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
* A plugin (CAP) for providing information about the context.
*
* @author <a href="mailto:emerson@labpesquisas.tci.ufal.br>Emerson Loureiro
*         </a>
*
* @version 1.0
*/
@att()
public abstract class ContextAwarenessPlugin extends FunctionalComponent {

  @behavior(
    abstract static public void registerContextListener(Application1 app){
      ContextDelivererThread contextDelivererThread = new ContextDelivererThread(app);
      contextDelivererThread.start();
    })
  public final void registerContextListener(ContextCondition condition,
    ContextEventListener listener) throws UnknownContextKeyException {
    if (this.checkConditionKey(condition.getContextInformationKey())) {
      ContextDelivererThread contextDelivererThread =
       new ContextDelivererThread(this, condition, listener);
      Hashtable listenersTable =
       (Hashtable) this.threadsTable.get(condition.getContextInformationKey());
      if (listenersTable == null) {
        listenersTable = new Hashtable();
        this.threadsTable.put(condition.getContextInformationKey(), listenersTable);
      }
```

```
      listenersTable.put(listener, contextDelivererThread);
      contextDelivererThread.start();
    } else { throw new UnknownContextKeyException(condition.getContextInformationKey()); }
  }
}

//-------------------------------------------------------------------------
package org.percomp.wings;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

/**
* Class for delivering the events about the context.
*
* @author <a href="mailto:emerson@labpesquisas.tci.ufal.br>Emerson Loureiro
*         </a>
*
* @version 1.0
*
*/
@att(Application1 app;)
class ContextDelivererThread extends Thread {

  @behavior(
    abstract public void run()
      loop(true) {
        if (BatteryLowCondition.isSatisfied()) {
          this.app.batteryWarnning = true;
          return;
        }
      }
    })
  public void run() {
    while (true) {
      String contextInformationKey = this.condition.getContextInformationKey();
      try {
        Object contextInformationValue =
         this.contextManager.retrieveContextInformation(contextInformationKey);
        if (this.condition.isSatisfied(contextInformationValue)) {
          ContextEvent event =
           new ContextEvent(contextInformationKey, contextInformationValue);
          listener.receiveContextEvent(event);
        }
      } catch (ContextInformationRetrievalException e) {this.listener.receiveError(e);}
      if (this.interruptedRequested) { return; }
      try { Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace(); }
    }
  }
}

//-------------------------------------------------------------------------
package org.percomp.wings;

import java.util.Collection;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;

import core.jaca.annotations.att;
import core.jaca.annotations.behavior;

import net.compor.frameworks.jcf.ComporComponentNotStartedException;
import net.compor.frameworks.jcf.ComporServiceInvocationException;
import net.compor.frameworks.jcf.Container;
import net.compor.frameworks.jcf.EventAnnouncement;
import net.compor.frameworks.jcf.ExecutionScript;
import net.compor.frameworks.jcf.ServiceRequest;
import net.compor.frameworks.jcf.service.ServiceResponse;

/**
 * A facade to the services provided by the middleware.
 *
 * @author <a href="mailto:emerson@labpesquisas.tci.ufal.br>Emerson Loureiro
 * </a>
 *
 * @version 1.0
 *
 */
@att(
  static MiddlewareFacade singletonInstance = new MiddlewareFacade();
  WingsExecutionScript script;)
```

```
public final class MiddlewareFacade {
  @behavior( return singletonInstance;)
  public static MiddlewareFacade getDefault() { return singletonInstance; }

  @behavior(task;)
  public void startMiddleware() {
    this.script.init();
  }

  @behavior(abstract public void addContextAwarenessPlugin(){ task; })
  public void addContextAwarenessPlugin(ContextAwarenessPlugin cap)
   throws CannotAddPluginException {
    this.script.addContextAwarenessPlugin(cap);
  }

  @behavior(
    abstract void registerContextListener(boolean boo, Application app){
      this.script.registerContextListener(boo,app);
    })
  public void registerContextListener(ContextCondition condition, ContextEventListener listener)
   throws UnknownContextKeyException, ContextInformationRetrievalException {
     try { this.script.registerContextListener(condition, listener);
     } catch (ComporServiceInvocationException e) {
       throw new ContextInformationRetrievalException(condition.getContextInformationKey(), e);
     } catch (ComporComponentNotStartedException e) {
       throw new ContextInformationRetrievalException(condition.getContextInformationKey(), e);
     }
   }

  @att()
  private class WingsExecutionScript extends ExecutionScript {

    @behavior(
    abstract void registerContextListener(boolean boo, Application app){
      Compor.publish(boo,app);
    })
    void registerContextListener(ContextCondition condition, ContextEventListener listener)
     throws ComporServiceInvocationException, ComporComponentNotStartedException,
     UnknownContextKeyException {
       String key = condition.getContextInformationKey();
       String serviceName = (String) this.registerContextListenerTable.get(key);
       if (serviceName != null) {
         Object[] parameters = new Object[] { condition, listener };
         ServiceRequest request = new ServiceRequest(serviceName, parameters);
         ServiceResponse response = this.exec(request);
         if (response.getException() != null) {
           throw (UnknownContextKeyException) response.getException();
         }
       }
     }
  }
}

//-------------------------------------------------------------------------
@att()
public class BatteryLowCondition implements ContextCondition {

  @behavior(
    abstract static boolean isSatisfied(){
      if (batterylevel < 1) return true;
      else return false;
    })
  public boolean isSatisfied(Object batteryLevel) {
    int level = ((Integer) batteryLevel).intValue();

    if (level <= 5) {
      return true;
    } else {
      return false;
    }
  }
}
```

Apêndice F

Instalação e uso do *plugin*

Para que determinado sistema possa ser verificado usando a técnica apresentada neste trabalho, deve-se instalar o *plugin* no ambiente de desenvolvimento Eclipse. Para isto, basta ao desenvolvedor extrair um arquivo *ZIP* dentro da pasta chamada *plugins* localizada no diretório raiz do Eclipse. Este arquivo é o *instalador* da ferramenta. Após iniciado o Eclipse, um menu adicional pode ser visto no ambiente. Tal menu é destacado na Figura F.1.

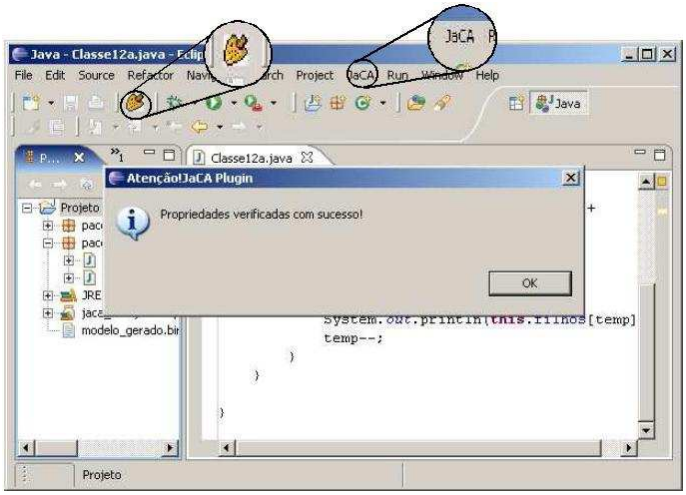

Figura F.1: Destaque ao menu inserido e botão do *plugin*.

Para que seja possível anotar o código com as anotações apresentadas no Capítulo 3, um arquivo JAR contendo as especificações das anotações deve ser adicionado ao *Java Build Path* do projeto que contém as classes a serem anotadas. Para cada projeto a ser verificado, o mesmo arquivo (ou cópia) deverá ser adicionado ao respectivo *Java Build Path*. Após instalado, o *plugin* é capaz de gerar o modelo comportamental de qualquer programa Java anotado com a linguagem JaCA.

Bibliografia

[All01] Agile Alliance. Manifesto for Agile Software Development. http://agilemanifesto.org/, 2001.

[AM91] A. W. Appel and D. B. MacQueen. Standard ML of new jersey. In Jan Maluszynski and Martin Wirsing, editors, *Proceedings of Programming Language Implementation and Logic Programming (PLILP '91)*, volume 528 of *LNCS*, pages 1–14, Berlin, Germany, aug 1991. Springer.

[Bei90] Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

[BJ01] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS01, Tools and Algorithms for the Construction and Analysis of Software*, number 2031 in LNCS, pages 299–312. Springer–Verlag, 2001.

[Blu01] Bluetooth SIG, Inc. Specification of the Bluetooth System, February 2001. Web document. URL `http://www.bluetooth.com/dev/specifications.asp` accessed January 21, 2002 02:45 UTC.

[BS00] Jennifer Bray and Charles F. Sturman. *Bluetooth: Connect Without Cables*. Prentice Hall, 1st edition, Dezembro 2000.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Lecture Notes in Computer Science: Logic of Programs*, volume 131. Springer-Verlag, 1981.

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraisi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In *Proc.*

11th Intl. Symp. on Comput. Hardware Description Lang. and their Applications, 1993.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CMM05] Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software refinement with perfect developer. In *SEFM*, pages 363–373, 2005.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–421. ACM Press, May 1999.

[DHJ+01a] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, Robby, C. S. Păsăreanu, W. Visser, and H. Zheng. Tool-Supported program abstraction for Finite-State verification. pages 177–187, Toronto, Canada, May12–19 2001. IEEE Computer Society.

[DHJ+01b] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, Robby, C. S. Păsăreanu, W. Visser, and H. Zheng. Tool-Supported program abstraction for Finite-State verification. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 177–187. IEEE Computer Society, 2001.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, October 1971. Reprinted in *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrot, Eds., Academic Press, 1972, pp. 72–93.

This paper introduces the classical synchronization problem of Dining Philosophers.

[DNS03] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories, July 2003.

[Ecl06] Eclipse. `http://www.eclipse.org`, 2006.

[Eri06] Ericsson. Tecnologia sem fio Bluetooth. `http://www.ericsson.com.br/bluetooth/index.asp`, 2006.

[FLL+02] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java, 2002.

[GPS96] Patrice Godefroid, Doron Peled, and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In Steven J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and analysis*, pages 261–269. ACM Press, 1996.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[Jam88] James S. Collofello. Introduction to Software Verification and Validation, December 1988. (ftp).

[Jen92] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1*. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1992.

[JLi06] JLint. Find Bugs in Java Programs. `http://jlint.sourceforge.net/`, 2006.

[Joh06] John D. Mitchell. Java Tip 106: Static inner classes for fun and profit. `http://www.javaworld.com/javaworld/javatips/jw-javatip106.html`, 2006.

[JPF05] JPF. Java PathFinder. `http://javapathfinder.sourceforge.net/`, 2005.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – a language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 14(2):175–211, Apr 1996.

[JUn06] JUnit. `http://www.junit.org`, 2006.

[Kat99] Joost-Pieter Katoen. *Concepts, Algorithms and Tools for Model Checking*, volume 32-1 of *Arbeitsberichte der Informatik*. Friedrich-Alexander-Universität Erlangen Nürnberg, 1999.

[KdB04] Jan Kettenis and Remco de Blok. Getting Started With Unit-testing, December 2004. Oracle Corporation.

[Kin06] Joe Kiniry. ESC/Java2. `http://secure.ucd.ie/products/opensource/ESCJava2/`, 2006.

[KMJ02] S. Khurshid, D. Marinov, and D. Jackson. An Analyzable Annotation Language, 2002.

[Kra99] Douglas Kramer. API Documentation from Source Code Comments: A Case Study of Javadoc. In *Proceedings of the 7th Annual International Conference of Computer Documentation (SIGDOC-99)*, pages 147–153. ACM Press, 1999.

[LBB+06] Emerson Loureiro, Frederico Bublitz, Nadia Barbosa, Hyggo Almeida, Angelo Perkusich, and Glauber Ferreira. A flexible middleware for service provision over heterogeneous pervasive networks. In *4th International Workshop on Mobile and Distributed Computing*, Niagara Falls, NY, USA, June 2006. IEEE Computer Society. Accepted for publication.

[LBR03] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, Apr 2003.

[Lim06] Escher Technologies Limited. Perfect language, 2006. `http://www.eschertech.com/product_documentation/`.

[LvH85] David Luckham and Friedrich W. von Henke. An overview of anna - a specification language for Ada. *IEEE Software*, 2(2):9–23, Mar 1985.

[Mai00] M. Maidl. The common fragment of CTL and LTL. In IEEE, editor, *41st Annual Symposium on Foundations of Computer Science: proceedings: 12–14 November, 2000, Redondo Beach, California*, pages 643–652, 2000. IEEE Computer Society Order Number PR00850.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM00] K. L. McMillan. The SMV System, 2000. Available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are Race Conditions? - Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), March 1992.

[NUn06] NUnit. `http://www.nunit.org/`, 2006.

[OM06] Elthon Allex Da Silva Oliveira and Eric Mercer. Comunicação pessoal entre membros dos laboratórios: GMF (LabPetri) da Universidade Federal de Campina Grande e *Bughunters Extraordinare Verification and Validation Laboratory* da *Brigham Young University*. Desde Janeiro, 2006.

[PG05] Joana Lúcio Paulo and João Graça. Javadoc(TM): ferramenta para comentar o código java. `http://po.tagus.ist.utl.pt/2005/apoio/IntroJavadoc.html`, 2005. Universidade Técnica de Lisboa.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, Oct 1977.

[pro06] Promela Language Reference, 2006.

[PyU06] PyUnit. `http://pyunit.sourceforge.net/`, 2006.

[RDF+05] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending jml for modular specification and verification of multi-threaded programs. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer, 2005.

[RDH04] R. Rodrguez, E. Dwyer, and M. Hatcliff. Checking strong specifications using an extensible software model checking framework, 2004.

[ROB00] ROBBY. Bandera specification language: A specification language for software model checking. Master's thesis, Kansas State University, 2000.

[Rob03] John Hatcliff Robby, Matthew B. Dwyer. Bogor: An extensible and highly-modular model checking framework. In *In the Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[RRDH04] Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking strong specifications using an extensible software model checking framework. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2004.

[RT05] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.1. Technical report, New York University – Computer Science Department, 2005.

[Sch04] Wolfgang Schmitt. Automated Unit Testing of Embedded ARM Applications, 2004. Hitex Development Tools.

[Sif90] J. Sifakis, editor. *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer, 1990.

[SM05] Daniel Aguiar Da Silva and Patrícia Duarte Machado. Geração de Objetivos de Testes para Sistemas Distribuídos Baseada em Técnicas de Verificação de Modelos. In *X Workshop de Teses em Engenharia de Software*, pages 33–38, 2005.

[SpE06] SpEx. Specifications Exploration. `http://spex.projects.cis.ksu.edu`, 2006.

[SPI] SPIN web site `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[Tec06] Escher Technologies. Perfect Developer. `http://www.eschertech.com/products/index.php`, 2006.

[Val98] Antti Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - Second Generation of a Java Model Checker, 2000.

[Wag05] Stefan Wagner. Towards software quality economics for defect-detection techniques, 2005.

[Wei91] Mark Weiser. The computer for the 21st centry. *Scientific American*, pages 94–104, September 1991.