MDA-Veritas: Uma Arquitetura MDA Estendida para Transformações de Sistemas Concorrentes Preservadoras de Semântica

Paulo Eduardo e Silva Barbosa

Tese de Doutorado

Orientadores:

Franklin de S. Ramalho

Jorge C. A. de Figueiredo

Coordenação do Curso de Pós-Graduação em Ciência da Computação

Universidade Federal de Campina Grande

Setembro-2011

B238m Barbosa, Paulo Eduardo e Silva

MDA-VERITAS: uma arquitetura MDA estendida para transformações de sistemas concorrentes preservadoras de semântica / Paulo Eduardo e Silva Barbosa. - Campina Grande, 2011.

213f: il. col.

Tese (Doutorado em Ciência da Computação), Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Orientadores: Prof. Dr. Franklin de S. Ramalho, Prof. Dr. Jorge C. A. de Figueiredo.

Referências.

1. Engenharia de Software. 2. Semântica Formal. 3. Transformações.
I. Título.

CDU 004.41 (043)

Agradecimentos

Tenho muito tempo para agradecer à minha família e amigos pela forma contínua que se fazem presentes em todas as etapas da minha vida. Contudo, quero registrar aqui que este trabalho não se concretizaria sem o apoio prestado por uma grande equipe. Inicialmente, agradeço aos meus orientadores Franklin Ramalho e Jorge Figueiredo pela concepção e lapidação da idéia, além do acompanhamento de todos os meus passos durante todos esses 5 anos de trabalho. Agradeço aos professores Luis Gomes e Anikó Costa pela forma com que acreditaram na idéia e me ajudaram a validá-la, prestando-me também um caloroso acolhimento. Agradeço aos estudantes Antônio Júnior, André Aranha e Antônio Avelino pelas longas horas dedicadas à construção dos pilares que concretizaram a idéia, sendo a base de sustentação nas horas das tribulações. Agradeço aos pesquisadores Filipe Moutinho e João Paulo Barros pelo interesse que tiveram em experimentar e contribuir com a consolidação da idéia. Além do mais, agradeço à projetista Uyara Travassos pelo apoio dado em todo o processo de organização do trabalho.

Finalmente, registro que esse trabalho também é fruto da ajuda dos amigos Rohit Gheyi, Tiago Massoni, Jacques Robin, Cássio Rodrigues, Dalton Guerrero, Rogério Rebelo, Rui Pais, Roberto Bittencourt, Anderson Ledo, Emanuela Cartaxo, Jemerson Damásio, Isabel Nunes, Ana Emília Barbosa, Andreza Vieira, Hermann Haeusler, Christiano Braga, Everton Galdino, Fábio do Carmo, Gustavo Rocha, além de vários outros integrantes do GMF da UFCG. Fica registrado aqui o meu muito obrigado!

Para fazer algo que antes ainda não estava lá, ... é decepcionante, porque os elementos separados já existiam e flutuavam através da história, mas nunca haviam sido juntados dessa maneira. Uni-los, montá-los, o novo sempre consistirá disso.

Anneke Kleppe parafraseando Connie Palmen.

Resumo

MDA é uma tendência de desenvolvimento de software que visa alterar o foco e os esforços dos modelos de desenvolvimento atuais. O método de implementação deixa de ser apenas a produção de código, e passa a também envolver modelos, metamodelos e transformações. Atualmente, essa abordagem tem sido diversificada com a inclusão de novos paradigmas que vão bem além do uso exclusivo dos padrões da OMG, como proposto originalmente. Contudo, a arquitetura MDA ainda sofre com a falta de formalização de alguns de seus artefatos e processos, levando a vários tipos de questionamentos. Um exemplo pertinente de questionamento se dá sobre o alto grau de ambigüidade dos modelos e transformações, originando problemas de baixa confiabilidade. Uma das conseqüências disso é o fato de que atualmente não existe uma maneira de garantir que transformações MDA sejam preservadoras de semântica, e nem que seus modelos envolvidos nas transformações sejam formais o suficiente para se permitir o uso de técnicas de verificação de equivalência, gerando críticas sobre a eficácia dessa abordagem. Esta tese de doutorado propõe lidar com esse problema, incorporando abordagens consolidadas de métodos formais na arquitetura MDA, tendo como contexto específico o desenvolvimento de software para sistemas embarcados com características de concorrência. Propomos extensões para parte da arquitetura MDA para que se possa construir modelos semânticos que representem aspectos estáticos e dinâmicos, ambos essenciais na semântica dos modelos envolvidos nas transformações e nos mecanismos de verificação de equivalência desses modelos. Com isso, obtemos a verificação de equivalência em transformações envolvendo modelos de sistemas concorrentes. Como avaliação do trabalho, provas de conceito, estudos de caso e avaliação experimental seguindo a abordagem GQM, envolvendo parcerias na academia e na indústria através de sistemas reais, foram implementados e avaliados. Verificamos equivalência entre modelos ao nível de transformações PIM-para-PIM, PSM-para-PSM e PIM-para-PSM como modelos de sistemas concorrentes descritos em redes de Petri e algumas de suas extensões.

Nota. Este projeto foi financiado pelo governo brasileiro através da agência CAPES e envolveu atividades de colaboração financiadas parcialmente pelo governo português através da agência FCT. 3337-5466

Abstract

MDA is a software development trend that aims to shift the focus and efforts of the current development methodologies. The implementation method changes from only code production to the usage of models, metamodels and transformations. Currently, this approach has been diversified with the inclusion of new paradigms that go beyond the only use of the MDA standards, as originally proposed. However, the MDA architecture still suffers from the lack of formalization of its artifacts and processes, leading to several sorts of questions. An important example of question is about the high ambiguity levels of models and transformations, originating problems of low reliability. One of the main consequences of this problem is the fact that still there is no way to ensure that MDA transformations are semantics preserving and neither the involved models are formal enough to allow the use of equivalence verification techniques, criticizing the effectiveness of this approach. This thesis proposes to deal with this problem by incorporating well consolidated formal methods techniques in the MDA architecture, having as specific context the software development for embedded systems with concurrent features. We propose extensions to part of the MDA architecture in order to construct semantic models to represent static and dynamic aspects, both essentials in the semantics of the involved models in the transformations and in the verification mechanisms of these models. With this, we achieve the verification of equivalence in transformations with models of concurrent systems. As evaluation of the work, conceptual proofs, case studies and an experimental evaluation following the GQM approach, involving partners in the academy and industry, were implmented and evaluated. We verify models equivalence at the level of PIM-to-PIM, PSM-to-PSM and PIM-to-PSM transformations with models of concurrent systems described and in Petri nets and some of its extensions.

Note. This project was supported by the brazilian government through the research agency CAPES and involved collaboration activities partially supported by the portuguese government through the research agency FCT.

Lista de Figuras

Lista de Tabelas

Conteúdo

1 Introdução

A Arquitetura Dirigida por Modelos (*Model-Driven Architecture* - MDA) [Miller and Mukerji, 2003, OMG, 2011a] é uma abordagem para desenvolvimento de software cuja ênfase se dá em modelos, metamodelos e transformações entre modelos. Temos que modelos são representações da estrutura, função ou comportamento de sistemas. Metamodelos são modelos que descrevem a estrutura dos modelos, provendo uma forma de definir linguagens para a construção de modelos ou programas de uma maneira sem ambigüidades. Transformações entre modelos são conjuntos de regras que descrevem como gerar um modelo de saída a partir de um ou vários modelos de entrada, através de consultas aos metamodelos. Desta forma, o papel das transformações também é chave na abordagem MDA, pois, além de serem o principal mecanismo para que um modelo construído torne-se algo executável, elas também auxiliam na modelagem, processando um sistema durante o seu ciclo de vida e tornando os modelos em elementos mais refinados e de maior funcionalidade.

MDA é uma realização das idéias presentes na MDE (Model-Driven Engineering) ou na MDSD (Model-Driven Software Development) [Bettin, 2004], porém, concebida especificamente pela OMG (Object Management Group) [OMG, 2011b]. Dois tipos de modelos clássicos na MDA, ou níveis de abstração, são: o PIM (*Platform Independent Model*) e o PSM (*Platform Specific Model*). O PIM apresenta uma descrição, completa ou não, do sistema, abstraindo detalhes de implementação e tem como exemplos algumas definições em UML (Unified Modeling Language) [OMG, 2011b], Alloy [Jackson, 2002] ou redes de Petri [Petri, 1962]. Por outro lado, o PSM anexa os conceitos específicos de uma plataforma, tendo como exemplos *beans* para serem implantados em servidores Java ou artefatos do framework *.Net*, concebidos a partir de transformações do modelos UML ou Alloy, ou ainda módulos com canais de comunicação descritos de acordo com alguma tecnologia, particionados a partir de modelos em redes de Petri.

Ainda logo após a concepção da MDA, a OMG definiu um padrão para construção de transformações, intitulado por QVT (Queries, Views and Transformations) [OMG, 2005]. Porém, devido à existência de soluções consolidadas ou à emergência de novas linguagens e ambientes eficazes, hoje temos diversos tipos de alternativas a esse padrão, e que, de fato, são bem mais utilizados. Alguns exemplos de frameworks com já um bom tempo de consolidação para esse propósito são o ATL-DT (Atlas Transformation Language Development Tools) [Bezivin et al., 2003], o ambiente AGG (Attributed Graph Grammar) [Ehrig et al., 2006a], a solução VIATRA (VIsual Automated

model TRAnsformations) [Csertan et al., 2002] ou a linguagem declarativa KerMeta Transformation Rule (KTR) [Falleri et al., 2006]. MDA define dois tipos de transformações: verticais e horizontais [OMG, 2011a, France and Bieman, 2001]. Em transformações horizontais, os modelos fonte e destino residem no mesmo nível de abstração, enquanto que nas transformações verticais, eles estão em níveis de abstração diferentes. Abstração e alguns tipos de refinamento de modelos são exemplos de transformações verticais, enquanto que refatoramento de modelos e outros tipos de refinamentos são exemplos de transformações horizontais [Kleppe and Warmer, 2003].

1.1 Contextualização

Na medida em que a complexidade dos sistemas aumenta, a construção do artefato transformação torna-se uma atividade cada vez mais elaborada e não trivial. Em diversos casos, perde-se completamente a noção sobre a confiabilidade deste artefato quando este se baseia apenas em conceitos puramente sintáticos. Uma ilustração interessante desse caso é dada em [Podnieks, 2005]. Este exemplo discute sobre a corretude de algumas transformações entre modelos UML para modelos relacionais RDBMS (Relational Database Management Systems), analisando algumas formas de implementação dessas transformações. Observa-se a ineficácia destas transformações devido à complexidade de atributos, ligações e questões de perda de informação, não permitindo reversão, comutatividade e preservação de significado das transformações. Outro trabalho que ilustra mais essa situação é [Rensink and Nederpel, 2008]. Nesse caso, os autores alertam para o problema da falta de semântica formal de QVT, e os problemas decorrentes disso em MDA, necessitando do uso de alguma teoria formal bem-consolidada para que se possa abordar essa questão.

De acordo com a análise que fizemos para os casos anteriores, constatamos, pela opinião dos próprios autores dos trabalhos, que faz-se necessário tratar modelos e transformações como alguma estrutura formal, e não apenas como construções simples (esquemas, por exemplo) que só servem de *templates* para instâncias. Modelos matemáticos devem ser formalizados utilizando alguma linguagem de primeira ordem, axiomas de lógica de predicados ou teorias particulares já consolidadas de acordo com o paradigma dos modelos envolvidos em tais questões. Isto deveria fazer com que os modelos fossem vistos como *interpretações* dos metamodelos, facilitando a aplicação prática da abordagem.

Um outro detalhe importante dessa análise é que as pesquisas direcionadas a esses questionamentos ainda não conseguiram chegar perto de uma solução unificadora, gerando mais perguntas

e dúvidas sobre esse domínio complexo. A única certeza que se tem, é que soluções puramente baseadas em sintaxe e que não possuam conceitos algébricos, mesmo que sejam baseada em técnicas sofisticadas como deduções, não poderão apresentar a garantia necessária de corretude para transformações MDA por conta do comportamento dinâmico dos modelos. Além do mais, como veremos, esse mesmo problema pode ser observado para outros casos, com transformações envolvendo modelos definidos em outros paradigmas.

Continuando a contextualização com mais um outro exemplo ilustrativo, consideremos um projeto cliente envolvido com transformações para o desenvolvimento de sistemas embarcados com fortes características do paradigma dos sistemas concorrentes, foco principal deste trabalho. Neste paradigma, tratamos de sistemas que possuem a propriedade de ter vários processos ou componentes executando ao mesmo tempo e que permitem interação de uns com os outros em vários momentos. Tomemos as linguagens redes de Petri e Máquinas de Estados Finitas conforme descritas pelos metamodelos simplificados presentes na Figura 1. Ao lado esquerdo, temos que uma rede de Petri é composta de vários elementos, que podem ser do tipo `Nó` ou do tipo `Arco`. Um nó, por sua vez, pode ser do tipo `Transição` ou `Lugar`. Além do mais, cada arco está associado a dois nós, um de chegada e um de saída. Já o metamodelo do lado direito define que uma máquina de estados é composta de vários estados, que possuem associação com uma transição de chegada e uma transição de saída. Desta forma, uma transformação comumente definida na prática por desenvolvedores em MDA em nosso grupo foi de transformar uma rede de Petri em uma máquina de estados finita através do mapeamento direto entre as seguintes metaclasses: (i) `RedeDePetri` para `MáquinaDeEstados`; (ii) `Transição` para `Transição`; e (iii) `Lugar` para `Estado`. O problema é que, apesar de estruturalmente haver correspondência entre esses elementos sintáticos de ambas as linguagens, comportamentalmente ambos diferem em algumas propriedades essenciais conforme descreve a Figura 2.

A Figura 2(i) compara o caso de transformação entre modelos sequenciais. Observamos que a máquina de estados gerada a partir da rede de Petri preserva o mesmo comportamento, mudando de estado após as execuções das transições `T1` em ambos os modelos. A Figura 2(ii) considera o caso de situações de conflito nos modelos. Percebemos que haverá uma escolha entre a execução das transições `T1` ou das transições `T2` em ambos os modelos. A Figura 2(iii) descreve um caso em que a transição `T1` do modelo redes de Petri apresenta a característica de concorrência. Neste caso, para que a máquina de estados preservasse essa característica, seria necessária a criação de

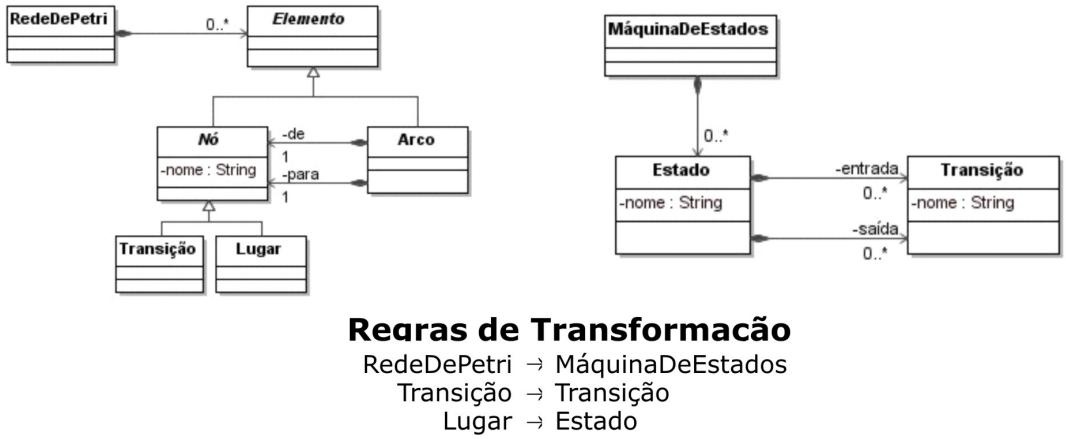

Figura 1: Transformação simples envolvendo Redes de Petri e Máquinas de Estados

duas transições `T1`, quebrando uma das regras da transformação que diz que cada transição rede de Petri deverá ser mapeada em exatamente uma transição máquina de estados. Finalmente, a Figura 2(iv) apresenta uma situação ainda mais crítica, onde a transição `T1` do modelo redes de Petri apresenta a característica de sincronização. Neste caso, temos uma situação totalmente não-determinística na transformação, onde não sabemos a que estados devemos associar a transição `T1` que será gerada na máquina de estados a ser produzida pela transformação. Este caso apresenta todas as fragilidades que podem ser ocasionadas por um método de construção de transformação baseado apenas em suposições sintáticas entre as linguagens que definem os modelos.

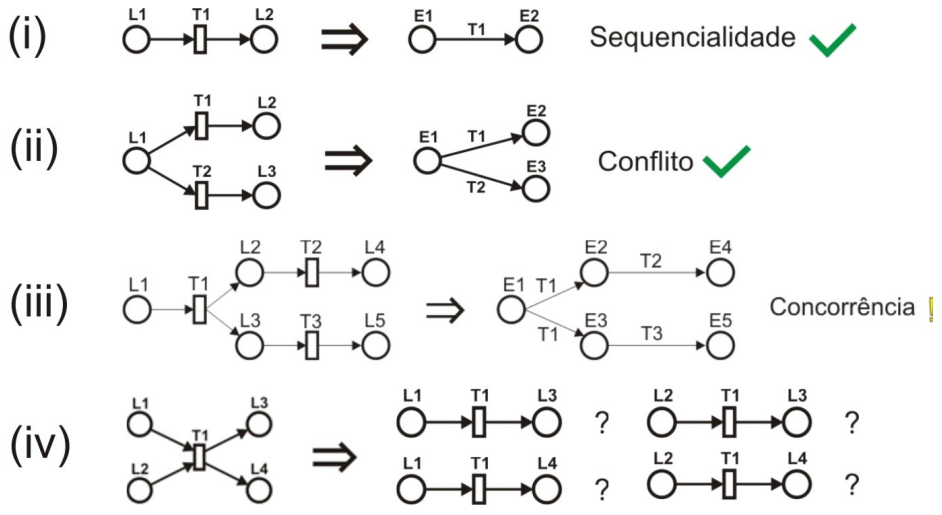

Figura 2: Casos de aplicação da transformação Redes de Petri e Máquinas de Estados

Apresentando um último exemplo motivador, consideremos uma transformação tradicional entre modelos descritos em redes de Petri para modelos descritos na linguagem Grafcet

[David and Alla, 1992]. Esta transformação MDA foi disponibilizada pelo site ATL Zoo [Atlantic Zoo, 2011]. Nela, um modelo rede de Petri é convertido para um modelo Grafcet, simplesmente atribuindo equivalências às construções sintáticas mais óbvias. Assim, uma instância da metaclasse raiz `PetriNet` é convertida para uma instância da metaclasse raiz `Grafcet`, e assim por diante para outras metaclasses, em um procedimento simples de acordo com a linguagem ATL [Bezivin et al., 2003]. O problema é que, para adeptos da comunidade de sistemas concorrentes, essa transformação não apresenta nenhuma confiabilidade. Alguns dos motivos principais são as diferenças entre os tipos de marcações de estado em cada linguagem, diferentes semânticas de execução, diferenças estruturais, tais como conflitos, onde transformações prévias de ajustes de equivalência deveriam ser aplicadas. Desta forma, há grandes possibilidades de os modelos de entrada e de saída dessa transformação apresentarem propriedades distintas, tais como situações de deadlock, por exemplo.

Vários outros questionamentos ainda ficam evidentes sobre a validade desta transformação devido à falta de garantia de equivalência entre esses modelos. A Figura 3, através de um diagrama de Venn, sumariza as classes e os conceitos conjuntos e disjuntos que modelos nessas linguagens (redes de Petri e Grafcet) apresentam [David, 1995]. Observamos que há modelos redes de Petri (β) que não podem possuir um modelo Grafcet equivalente e vice-versa (γ). Primeiramente, temos que uma dada marcação em redes de Petri é um inteiro maior que zero, enquanto que em Grafcet as marcações são valores verdade. Além do mais, em redes de Petri temos que as transições são disparadas uma de cada vez, respeitando a atomicidade de cada ação, enquanto que no Grafcet todas as transições habilitadas são executadas instantâneamente, produzindo uma diferença crucial entre as semânticas de execução de cada linguagem. No caso de β, a rede de Petri apresenta propriedades impossíveis de serem reproduzidas nos grafcets, tal como o não-determinismo. Já no caso de γ faz-se necessária a aplicação de uma transformação para que se tornem adequados a serem reproduzidos como redes de Petri. Isso não é detectado pela transformação, produzindo modelos errôneos em sua saída.

Sabemos que nem todas as transformações possuirão um estudo detalhado que esclareça as peculiaridades dos domínios semânticos de cada modelo envolvido, como acontece no caso anterior. A necessidade de um trabalho desta natureza justifica-se pelo fato de haver diversos casos de transformações MDA que não possuem quaisquer indícios de equivalência entre os modelos e nem trabalhos desta natureza que estabeleçam as relações de equivalência entre os domínios semânticos

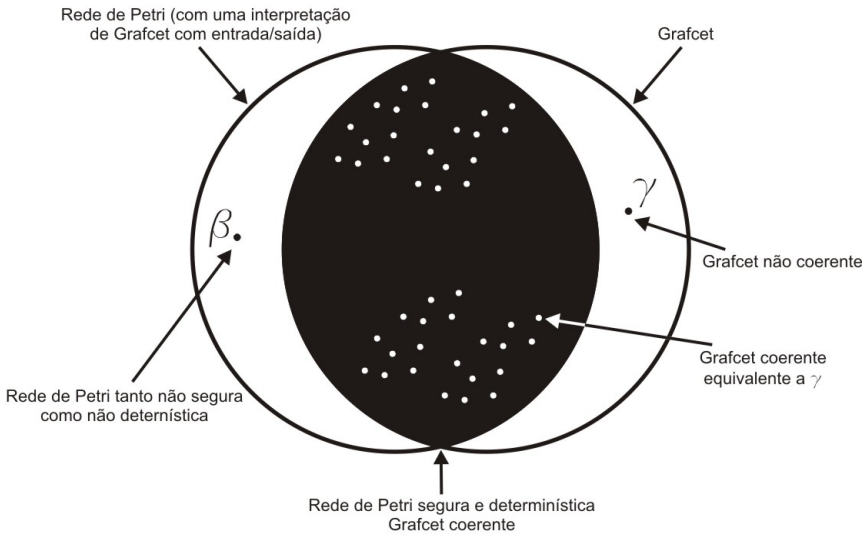

Figura 3: Equivalência entre modelos redes de Petri e Grafcet

em questão, que poderiam servir de alertas e guias para os usuários finais.

Desta forma, a maioria das linhas de pesquisa em preservação de semântica das transformações, abordam melhorias na construção desse artefato, tais como novas ferramentas de suporte, abordagens de modelagem da própria transformação, novas construções nas linguagens de transformação e novos paradigmas. Contudo, apesar dos benefícios trazidos por esses esforços, percebemos um foco dado de forma demasiada ao artefato transformação, e a tentativa de generalização por especificações válidas para o maior número de cenários possíveis, acabou por obscurecer o papel que o comportamento específico dos modelos envolvidos nas transformações exercem sobre esta complexidade.

1.2 Definição do Problema e Justificativa

Este trabalho concentra-se no problema da falta de garantia de equivalência semântica entre modelos envolvidos nas transformações MDA existentes. Isto justifica-se pelo fato de que, embora MDA já se apresente como uma peça importante para a engenharia de sistemas, ainda não foram especificadas maneiras para garantir que suas transformações, de fato, promovam o uso dos modelos de forma confiável durante todo o ciclo de desenvolvimento de um sistema. Particularmente, a falta de artefatos que permitam uma representação formal dos elementos envolvidos leva a situações indesejadas de ambigüidade e de falta de confiabilidade quando comparando as estruturas e os comportamentos dos modelos de entrada e de saída das transformações. Além das caracterís-

ticas semânticas serem muito mais difíceis de definir e descrever do que as puramente sintáticas [Schmidt, 1986], ainda não existe um método padrão para escrita dessa semântica formal e nem ferramentas adequadas para o seu processamento. De fato, tudo que a sintaxe não é capaz de descrever é atribuído ao domínio semântico, não havendo uma linha tão clara e nem fixa entre esses dois domínios na MDA.

1.3 Objetivos do Trabalho

O objetivo geral deste trabalho é investigar uma forma de verificar equivalência entre modelos envolvidos em transformações no framework MDA. O intuito é realizar isto através do reuso de técnicas consolidadas para métodos formais, além de promover o máximo de reuso possível das linguagens, padrões e ferramentas que já venham recebendo uma boa adoção pela comunidade MDA.

Detalhando em objetivos mais específicos, objetivamos descrever conceitos da semântica formal e de outras técnicas formais mais através de uma extensão formal do modelo de arquitetura MDA existente atualmente. Desta forma, proveremos a agregação de metamodelos semânticos para representação de conceitos dos domínios semânticos das linguagens envolvidas e uma forma de derivação automática de modelos semânticos de acordo com esses domínios semânticos. Além disso, objetivamos o estabelecimento de um mecanismo de verificação formal de equivalência de comportamento entre modelos semânticos que seja de fácil manuseio e entendimento por parte do cliente da solução.

1.4 Escopo do Trabalho

Abordamos, como principal cenário de realização do trabalho, o contexto que envolve modelos definidos no paradigma concorrente presentes em transformações MDA. Os cenários de aplicação e estudos de caso envolvem a idéia de desenvolvimento de sistemas baseado em modelos para sistemas embarcados, que recentemente vem recebendo uma atenção especial da comunidade científica [Gomes et al., 2005]. Esse cenário apresenta diversas transformações com característica fortes de concorrência. Em especial, nos interessamos por abordagens baseadas em linguagens formais, tais como redes de Petri ou ainda novas linguagens específicas de domínio, como as IOPTs [Gomes et al., 2007]. A partir dessas linguagens, nos concentramos em verificar e avaliar transformações específicas empregando modelos PIMs e PSMs.

A transformação MDA abordada é a *Splitting* [Gomes and Costa, 2007]. Ela está inserida no processo de desenvolvimento de sistemas embarcados do Projeto FORDESIGN [Gomes et al., 2005], resumido na Figura 4, que enfatiza técnicas de co-projeto de hardware e de software em abordagens baseadas em modelos. Nesse contexto, técnicas automáticas em verificação de propriedades são requisitadas no intuito de avaliar a transformação ao garantir a equivalência semântica entre o modelo de sistema inicial e os componentes particionados gerados.

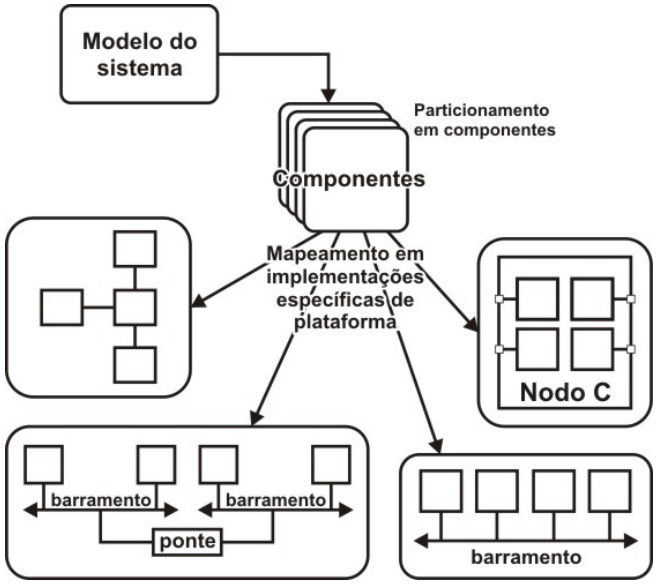

Figura 4: Particionamento e mapeamento de modelos

1.5 Contribuições Esperadas

A satisfação da necessidade de se determinar *se dois sistemas apresentam propriedades semelhantes* após a aplicação de transformações MDA no domínio de sistemas concorrentes deverá viabilizar soluções para diversos problemas comuns da engenharia de sistemas, tais como refatoramento e refinamento de modelos, geração de código, engenharia reversa, particionamento e junções durante o ciclo de vida e outras técnicas que já venham sendo comumente aplicadas nessa infraestrutura [Kleppe and Warmer, 2003]. Para o domínio de projetos envolvendo sistemas embarcados com características de concorrência, que é o alvo de nossa investigação, temos as seguintes contribuições:

- *Incorporar técnicas formais à MDA*. Embora MDA seja uma tendência para o desenvolvimento de sistemas de sistemas, muito da sua confiabilidade sempre foi questionada devido

à ausência de um suporte formal às idéias apresentadas por esse framework de desenvolvimento. Esperamos que, com esse trabalho, novas idéias sejam levantadas pela comunidade MDA em relação à preservação de semântica em transformações. Essa contribuição se materializa na extensão da arquitetura MDA denominada por MDA-Veritas (VERIficadora de Transformações MDA Assistida por Semântica), com o objetivo de incorporar verificação de propriedades da semântica formal nos modelos envolvidos em suas transformações.

- *Melhorar confiabilidade na manipulação e processamento dos modelos*. Modelos PIMs e PSMs são constantemente transformados, refatorados ou refinados com o objetivo de decomposição na direção de que atividades específicas possam ser delegadas também para plataformas específicas. A prova de uma relação de equivalência entre modelos permitirá a aplicação dessas operações sem alguns tipos de danos que são esperados constantemente, tais como: perda ou adição de informação desnecessária, erros de sincronização entre os fragmentos do modelo em caso de operações de decomposição ou de junção, alteração no conjunto de estados e caminhos possíveis referentes ao grafo de ocorrência de cada modelo [Jensen, 1986], mudanças nas propriedades comportamentais satisfeitas pelos modelos e outros fatores mais. Além do mais, a equivalência na satisfação dessas propriedades permitirá garantir que uma transformação MDA seja preservadora de semântica para as instâncias verificadas.

- *Melhorar confiabilidade na geração de código executável*. Modelos PSMs passam por transformações para construções imperativas a serem executadas em diversas plataformas de sistemas concorrentes. A formalização do comportamento desses modelos permitirá a representação correta dessas construções em linguagens específicas [Gomes and Costa, 2006], como por exemplo ANSI C, System C ou VERILOG, para diversos tipos de plataformas.

- *Aumentar o grau de representatividade semântica de plataformas de execução*. Modelos PSMs gerados para diversas plataformas poderão ter seu comportamento representado de acordo com as regras que regem a plataforma específica de execução. Como se sabe, essas plataformas podem apresentar restrições quanto ao uso de recursos, desempenho e outros fatores mais. A representação de propriedades semânticas nessas plataformas enunciará um utilidade a mais para os modelos PSMs, trazendo benefícios imediatos tais como redução de custos na confecção dessas plataformas específicas, economia de recursos, aumento de

confiabilidade e outros mais.

- *Permitir uma infraestrutura para verificação de preservação de semântica em MDA*. Este trabalho possibilitará um conjunto de ferramentas integrado que permita execução e análises com um certo grau de automação.

- *Prover uma avaliação completa para o paradigma concorrente através de modelos em redes de Petri e redes IOPT*. Através do método de avaliação GQM, abordaremos a solução proposta em níveis de medição de acordo com objetivos definidos para este domínio, caracterizarmos a eficácia da solução e do seu método de avaliação, além de formalizar um conjunto de métricas que nos ajudem a responder precisamente sobre comportamentos nas situações investigadas.

1.6 Estrutura da Tese

Esta tese compreende mais 7 capítulos. O Capítulo 2 apresenta os principais conceitos necessários para o entendimento do conteúdo proposto. O Capítulo 3 descreve a extensão da arquitetura MDA (MDA-Veritas) para propósitos de verificação de equivalência entre modelos. O Capítulo 4 apresenta a instanciação da MDA-Veritas para um cenário de verificação envolvendo modelos de sistemas concorrentes. No Capítulo 5, transformações são aplicadas referindo-se a modelos PIMs e PSMs no contexto de sistemas embarcados. O Capítulo 6 apresenta uma avaliação experimental da instanciação e das verificações realizadas na MDA-Veritas até então. O Capítulo 7 discute e avalia uma quantidade de abordagens que possuem alguma similaridade com este trabalho. Finalmente, o Capítulo 8 faz as considerações finais, apresenta as contribuições e indica tendências de trabalhos futuros.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos principais necessários para o entendimento do restante dessa tese. A Seção 2.1 inicia-se com a descrição, de forma genérica, da arquitetura MDA. Esta seção termina tratando do artefato transformação entre modelos, inclusive com sua linguagem de descrição, constituindo uma peça fundamental de análise deste trabalho. A Seção 2.2 descreve o principal domínio de desenvolvimento de aplicações abordadas por nossa solução, que são os sistemas embarcados com características de concorrência, quando usando abordagens dirigidas por modelos no desenvolvimento de software e de hardware. A Seção 2.3 apresenta as linguagens de redes de Petri e sua extensão IOPT. Essas linguagens abordam características das linguagens de modelagem de sistemas embarcados escolhidas para representação dos modelos PIM e PSM envolvidos nas transformações de sistemas concorrentes e que serão verificados. A Seção 2.4 descreve alguns dos formalismos existentes para definição de semântica de linguagens de programação e de modelagem que serão úteis para esse trabalho. Esta seção também apresenta o formalismo da lógica de reescrita, com sua forma unificadora de representar conceitos da semântica formal no domínio de sistemas concorrentes, descreve a técnica de representar semântica algebricamente, usando o poder da lógica de reescrita para unificar as visões apresentadas anteriormente e a teoria das categorias, com uma forma abstrata de lidar com esses conceitos algébricos. Ao final, a seção apresenta a técnica de model-checking, associando estados e suas mudanças nos estados dos modelos, para comparações de comportamentos entre modelos envolvidos em transformações. Por fim, a Seção 2.5 apresenta uma breve recapitulação com análise dos conceitos apresentados.

2.1 Arquitetura Dirigida por Modelos

A Arquitetura Dirigida por Modelos (*Model-Driven Architecture*) OMG [OMG, 2011a], ou MDA, define uma abordagem na qual separa, para um sistema, sua especificação de funcionalidade de qualquer plataforma tecnológica específica. Um diferencial de MDA reside na natureza dos artefatos que são produzidos durante o processo de desenvolvimento. Modelos são considerados entidades de primeira classe.

Em MDA, uma aplicação pode ser descrita por vários modelos em níveis de abstrações diferentes. Eles estão desenhados na Figura 5, onde podem ser transformados em outros tipos de modelos e são classificados como a seguir. Para ilustrar esta classificação, consideramos, com propósitos

ilustrativos, o contexto de modelagem até a geração de código de um sistema concorrente qualquer.

- Modelo Independente de Computação (*CIM - Computation Independent Model*): é independente de estrutura e processamento. Geralmente eles são mais orientados às regras de negócio do sistema. Por exemplo, no domínio de sistemas concorrentes, um conjunto de workflows concorrentes poderia ser considerado uma notação adequada para representar um CIM.

- Modelo Independente de Plataforma (*PIM - Platform Independent Model*): é independente de qualquer tecnologia de implementação específica. Considera-se este modelo como os requisitos e o projeto do sistema. Por exemplo, no domínio de sistemas concorrentes, um modelo em redes de Petri representando diversas características de concorrência do sistema, como sincronização, poderia ser considerado uma notação adequada para representar um PIM. MDA enfatiza que o projeto correto de um PIM deverá sobreviver às mudanças tecnológicas nas quais esse modelo será submetido.

- Modelo Específico de Plataforma (*PSM - Platform Specific Model*): especifica o sistema em termos das estruturas de implementação que estejam disponíveis em uma dada tecnologia. Por exemplo, no domínio de sistemas concorrentes, um modelo representando diversas características de concorrência do sistema, mas já elaborado por construções sintáticas de alguma plataforma específica, como uma linguagem imperativa ou outro caso, faria o papel de um PSM.

- Código: corresponde à representação de um PSM em código fonte (sintaxe concreta), permitindo sua execução direta em alguma plataforma. Geralmente, são representados em alguma linguagem de programação, arquivos binários ou executáveis. Por exemplo, no domínio de sistemas concorrentes, um código VHDL a ser executado em uma plataforma FPGA com semântica de execução paralela faria o papel de código.

MDA também especifica que os modelos devem ser especificados também por outros modelos, que são chamados de metamodelos e representam as regras para que modelos em uma certa linguagem sejam bem formados sintaticamente. Por exemplo, os metamodelos de redes de Petri ou de uma linguagem imperativa como C, contemplam todas as definições apresentadas pela gramática dessas linguagens, definindo de uma forma gráfica ou textual a sintaxe abstrata de uma linguagem.

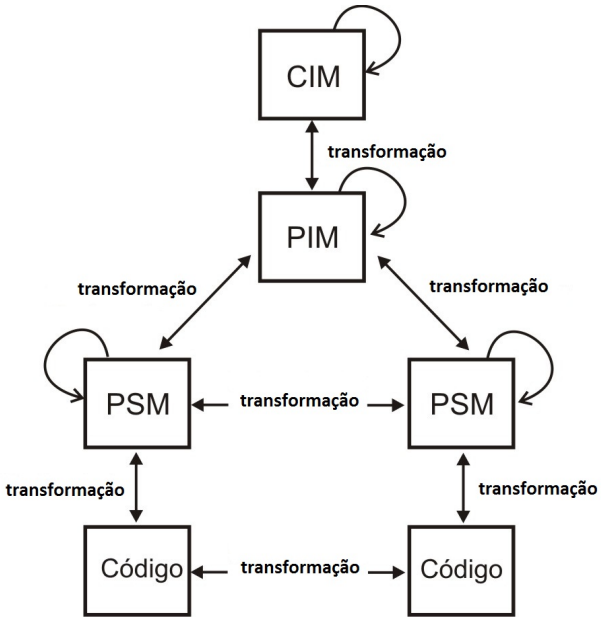

Figura 5: Relação de Tranformações entre os Modelos

Além do mais, um metamodelo deve ser definido por um meta-metamodelo, que seria a definição de uma linguagem que permitisse descrever qualquer construção de linguagem, como acontece com a notação EBNF [Aho et al., 1988] no caso de gramáticas. A partir daí, este meta-metamodelo deve se auto-definir, ou seja, reusar seus próprios conceitos.

Além da definição dos modelos como entidades, costuma-se definir operações dedicadas a esses modelos. A operação de maior evidência em MDA é a de transformações entre modelos. Transformação entre modelos é o processo de converter um modelo de origem, que está conforme com um metamodelo de origem, em um modelo destino, que também esteja conforme com um metamodelo de destino [Kleppe et al., 2003]. Os modelos da Figura 5 relacionam-se através de transformações, onde podem ser distinguidas como transformações verticais e horizontais entre os vários tipos de modelos, e transformações do modelo para um outro do mesmo tipo, representadas pelos auto-laços.

Dentro do contexto de MDA, há um padrão para especificação de transformações em modelos: MOF-QVT [OMG, 2005]. Há algumas linguagens que apresentam alguma compatibilidade com esse padrão. Entre elas, a mais popular é denominada por ATL [Bezivin et al., 2003], que será descrita na próxima seção. Estas transformações podem também combinar suas linguagens com especificações de consultas nos modelos e metamodelos através do reuso da linguagem OCL [Clark and Warmer, 2002].

A arquitetura MDA [OMG, 2011b], que sumariza as idéias levantadas anteriormente, é apresentada na Figura 6, que mostra o contexto de dois modelos envolvidos em uma transformação: o modelo de entrada (no lado esquerdo) e o modelo de saída (no lado direito). Ela possui quatro camadas: M0, M1, M2 e M3. A camada M0, que fica na parte mais baixa da figura, descreve a sintaxe concreta de um dado modelo. A camada M1 incorpora artefatos que representam abstrações em um modelo. A camada M2 descreve o metamodelo que serve como uma gramática para descrever e checar a corretude da sintaxe do modelo desenvolvido na camada M1. No topo, a camada M3 descreve a camada M2 usando MOF [OMG, 2011b] (*Meta Object Facility*), que é a linguagem específica para descrição de metamodelos provida pela OMG. Como MOF se auto-descreve, não requisita mais metamodelos e níveis acima. Finalmente, as transformações entre modelos permitem a geração automática de modelos de saída a partir de modelos de entrada na camada M1. Elas são definidas em termos de descrições de metamodelo e lidam apenas com aspectos sintáticos.

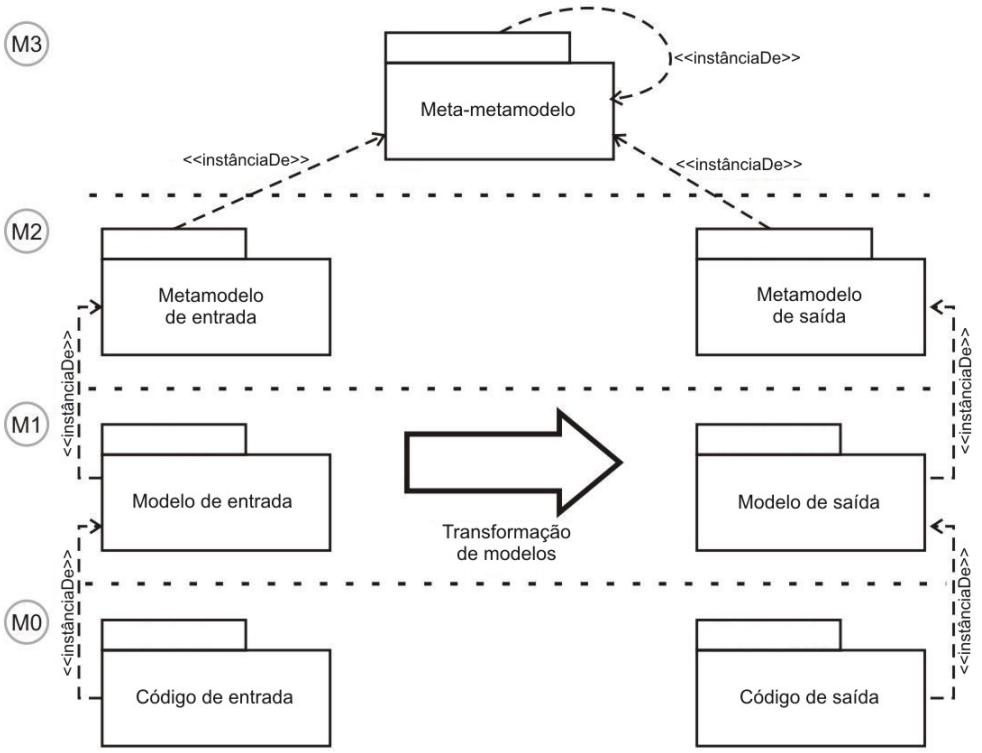

Figura 6: Arquitetura MDA de Quatro Camadas

ATL - Atlas Transformation Language

A ATLAS Transformation Language (ATL) [Bezivin et al., 2003] é uma linguagem de modelagem específica de domínio para representar transformações de um conjunto de modelos fonte

em modelos destino, descritos por um ou vários metamodelos. Isto é feito com um estilo de programação híbrido que mistura o estilo imperativo e declarativo. Além do mais, justifica-se o uso de ATL pelo conjunto de ferramentas disponíveis e por existir um conjunto de transformações interessantes já disponíveis com essa ferramenta [Atlantic Zoo, 2011].

Para ilustrar a definição de uma transformação ATL, recorremos ao exemplo de transformação mencionado na introdução desta tese, na Seção 1.1, e que tem seu código apresentado a seguir, no Código 2.1.

Código 2.1 em ATL. Exemplo de uma transformação

```
1: module PetriNet2Grafcet;
2: create OUT : Grafcet from IN : PetriNet;
3: ...
4: rule Grafcet {
5:  from p : PetriNet!PetriNet
6:  to g : Grafcet!Grafcet
7:  (
8:    location <- p.location, name <- p.name,
9:    elements <- p.elements, connections <- p.arcs
10: )
11:}
```

Por definição, temos que transformações ATL são definidas usando *módulos*. Um módulo contém uma seção *header* (cabeçalho) mandatória, uma seção *import* (importadora), um número de *helpers* (ajudantes) e regras de transformação que definem a funcionalidade da transformação. A seção *header* dá o nome do módulo de transformação e declara os modelos fonte e destino. Os modelos fonte e destino são tipados por seus metamodelos. Assim, a linha 1 do Código 2.1 representa a declaração do módulo que contém a transformação rede de Petri para Grafcet. A linha 2 declara o modelo a ser criado (OUT) a partir de um modelo já existente (IN). A partir daí, a transformação será constituída por regras. Em uma regra, a seção `from` define qual entidade do modelo fonte será transformada em qual entidade do modelo destino, definido na seção `to`. Após a declaração da regra na linha 4, a linha 5 representa a seção `from`, partindo do metamodelo `PetriNet` e da metaclasse `PetriNet`, e a linha 6 representa a seção `to`, com destino ao metamodelo `Grafcet` e para a metaclasse `Grafcet`. O corpo da seção (linhas 7 a 10) define o significado da regra de transformação: os atributos `location`, `name`, `elements` e `arcs` de `Grafcet` são atribuídos a `location`, `name`, `elements` e `connections` de `PetriNet` respectivamente.

Duas seções adicionais podem ainda aparecer opcionalmente na regra: a seção `using`, que

define variáveis locais a serem usadas, e a seção `do`, que permite o uso de código imperativo. Além do mais, regras podem tanto ser casadas como chamadas. Uma regra casada segue uma abordagem declarativa, ou seja, ela será executada sempre que existam entidades no modelo fonte que casem com sua seção `from`, e uma regra chamada poderá ser executada apenas se for invocada por outra regra.

2.2 Projeto de Sistemas Embarcados

Um sistema embarcado é uma combinação de hardware e software computacionais projetados para executar uma função dedicada. Além do mais, em alguns casos, os sistemas embarcados são partes de um sistema ou produto maior [Barr, 2011]. Geralmente, eles são construídos para controlar um dado ambiente físico onde tem-se disponíveis dispositivos eletrônicos, sensores e atuadores operando de maneira conjunta.

De acordo com [Costa, 2010], são características essenciais de sistemas embarcados: (i) serem desenvolvidos para aplicações específicas; (ii) trabalhar continuamente; (iii) manter interação com o ambiente; (iv) precisarem de especificação e desenvolvimento corretos; e (v) serem caracterizados como reativos e com capacidade de processamento de tempo real.

Para o projeto do software desses dispositivos, a linguagem de especificação deve disponibilizar as seguintes características: (i) permitir uma computação apropriada dos modelos; (ii) ser portável e flexível; (iii) ser legível; (iv) ser executável; (v) permitir decomposição hierarquias, tanto ao nível estrutural como comportamental; (vi) ter um comportamento orientado a estados; (vii) permitir captura de eventos; (viii) permitir concorrência; e (ix) permitir sincronização, assincronia implícita e explícita e comunicação.

Recentemente, houve uma mudança na filosofia de desenvolvimento desses sistemas. A idéia era de preencher o vazio existente nas abstrações a nível de sistema e introduzir metodologias que levassem em consideração o projeto de hardware e software ao mesmo tempo. Nesse cenário, temos o desenvolvimento baseado em modelos (MBD), onde o desenvolvimento é dividido em termos de modelos de processos e de produto, representando as atividades de desenvolvimento e as entidades respectivamente. Desta forma, no contexto do desenvolvimento de sistemas embarcados, o conceito de modelos independentes e específicos de plataforma estarão sempre presentes, podendo assim, instanciar-se a promissora filosofia MBD com o framework MDA.

2.3 Redes de Petri e Redes IOPT

Diversos formalismos já provaram sua adequação ao projeto de sistemas embarcados de acordo com uma atitude orientada a modelos. Entre eles, enfatizamos o uso de formalismos baseados em controle, onde sistemas baseados em estados desempenham um papel principal, com diagramas de estados, diagramas hierárquicos, de seqüências e redes de Petri.

Redes de Petri são ferramentas gráficas e matemáticas para modelagem de sistemas. Devido às suas características, as redes de Petri são fortemente recomendadas para modelagem de sistemas com um alto índice de paralelismo ou concorrência, revelando informações importantes sobre a estrutura e o comportamento dinâmico do sistema modelado. Existe concorrência em um sistema, quando este emprega diversos agentes independentes entre si para computar e compartilhar recursos, além de existir uma infra-estrutura para a comunicação entre esses agentes. O conceito de sistemas concorrentes abrange várias arquiteturas computacionais, como por exemplo, desde arquiteturas fracamente e fortemente acopladas até largamente distribuídas, com características síncronas ou assíncronas.

A rede na Figura 7 identifica uma situação comum de modelagem de um sistema real, e é um bom exemplo para apresentarmos a notação gráfica das redes de Petri. De acordo com a definição, que será apresentada logo mais adiante, esta rede de Petri contém dois lugares (`Passo` e `Tempo`), representados por círculos, e apenas uma transição (`Contador`), representada pelo retângulo preto. A rede modela a existência de passos temporais, através de uma marca que sempre volta para `Passo`, e o armazenamento disso em unidades de tempo, através das marcas em `Tempo`. Esses modelos de relógio estão presentes em várias situações de modelagem de sistemas concorrentes. Observe que este modelo pode apresentar um comportamento infinito, pois o lugar `Tempo` terá tantas marcas quantas unidades de tempo esse sistema funcionará. Para limitarmos esse modelo, pode-se adicionar um recurso ao modelo chamado de *capacity* de um lugar [Desel et al., 2001]. Neste caso, o lugar `Tempo` teria uma capacidade de valor de 59 tokens associada a ele. Cada vez que esse número fosse ultrapassado, o número de tokens retornaria ao valor zero.

Já a rede na Figura 8 não consegue impor uma representação ao sistema sendo modelado de uma maneira tão satisfatória. Tenta-se modelar um sistema de controle de luminosidade, onde o lugar `waitingDark` espera por um acionamento, um sinal de escuridão, para que possa acender uma lâmpada, passando para o lugar `light`. O problema é que as transições existentes (`OFF`,

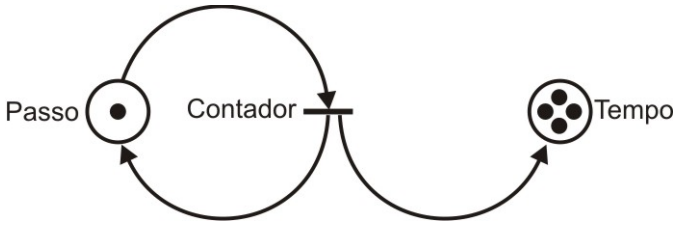

Figura 7: Exemplo de uma rede autônoma simples como um relógio

`BRIGHT` e `DARK`) não podem ser estimuladas pelo fato da rede ser autônoma, inibindo o uso de sensores, atuadores ou outros quaisquer. O sistema passa a ter um comportamento completamente ad-hoc.

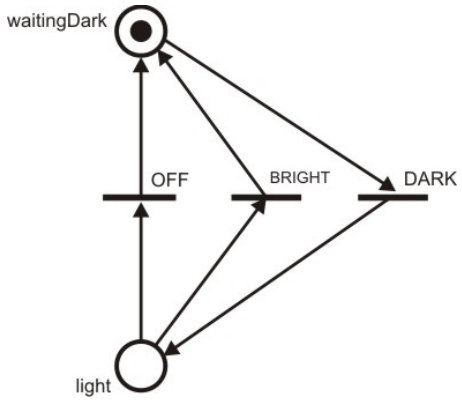

Figura 8: Exemplo de uma rede autônoma para um sistema de luminosidade

Definição 2.1 *Uma Rede de Petri N= ($\mathcal{P}$, $\mathcal{T}$, $\bullet T$, $T\bullet$) é definida por:*

- *Um conjunto finito $\mathcal{P}$ de lugares;*
- *Um conjunto finito $\mathcal{T}$ de transições;*
- *Uma função de entrada $\bullet T$: $\mathcal{T} \rightarrow 2^{\mathcal{P}}$;*
- *Uma função de saída $T\bullet$: $\mathcal{T} \rightarrow 2^{\mathcal{P}}$.*

Graficamente, nas redes de Petri autônomas, onde elementos não interagem com o ambiente, lugares são representados por círculos e as transições por barras ou retângulos. Uma rede de Petri pode ser representada por um grafo dirigido bipartido G = (V,E), com $V = \mathcal{P} \cup \mathcal{T}$ e $\mathcal{P} \cap \mathcal{T} = \emptyset$. Qualquer aresta e em E é incidente em um membro de $\mathcal{P}$ e um membro de $\mathcal{T}$.

Seja μ uma função μ: $\mathcal{P} \rightarrow \mathbb{N}$ que mapeia o conjunto de lugares $\mathcal{P}$ em números inteiros não negativos. Cada lugar armazena uma certa quantidade de marcas que é dada por esse número

inteiro. Denomina-se marcação numa rede de Petri a tupla M = $\langle\ \mu(p_1)$, $\mu(p_2)$, ... ,$\mu(p_L)\ \rangle$, com $\mu(p_i) \in \mathbb{N}$ e L = número de lugares.

Uma *rede de Petri marcada* é RM = (R, M), sendo que pelo menos a marcação de um lugar dessa rede seja diferente de zero, ou seja, $\mu(pi) \neq 0$. Na sua representação gráfica, as marcas são representadas por pontos nos lugares.

Cada rede possui uma configuração inicial que complementa a descrição do sistema. Com relação à dinâmica das redes de Petri, dizemos que uma transição está *habilitada* quando todos os seus lugares de entrada estiverem marcados com um número maior ou igual de marcas requisitadas pelos arcos. Uma transição *dispara* sempre que estiver habilitada e for a próxima a evoluir no sistema. O disparo de uma transição resulta em uma nova marcação da rede, retirando uma marca de cada lugar de entrada da transição que disparou e as colocando em cada lugar de saída.

Por redes de Petri representarem sistemas de controle, o conceito de espaços de estados é fundamental em seu uso. Uma representação em espaço de estados é um modelo matemático de um sistema composto de um conjunto de estados relacionados entre si por meio de transições. A representação em espaço de estados fornece uma maneira prática e compacta para modelar e analisar sistemas com múltiplas entradas e saídas. Essas estruturas são muito importantes no processo de verificação desses sistemas.

As redes de Petri podem trazer inúmeras vantagens para a definição de transformações preservadoras de semântica devido à sua forte definição matemática e sua semântica formal bem definida. Desta forma, elas se adequam à filosofia MDA por ser possível definir transformações de modelos automáticas ou semi-automáticas para que se possa trabalhar com os modelos em diferentes níveis de abstração. Além do mais, devido às vantagens apresentadas, poderemos obter poderosas técnicas de verificação para este domínio. Finalmente, redes de Petri dispõem de uma notação padronizada textual chamada de PNML (Petri Net Markup Language) [Weber and Kindler, 2003], que provê, em forma de metamodelo, um conjunto de ferramentas prontas para serem utilizadas pela comunidade MDA. O objetivo principal do PNML é prover interoperabilidade entre ferramentas de redes de Petri. Graças a esse formato, é possível trocar modelos, de acordo especificações padrão. O PNML framework provê as transformações, a partir de um modelo editado em redes de Petri, para um modelo objeto PNML e também para uma sintaxe textual PNML.

Embora as redes de Petri apresentem diversas vantagens para o engenheiro por suas características para modelar sistemas embarcados, ainda não existia uma noção de interação física com o

sistema. Nesse sentido, as IOPTs (Redes Lugar/Transição com Entrada e Saída) incluem conceitos como sinais e eventos, que permitem representar uma interação contínua com o ambiente. Dessa forma, elas estão classificadas na categoria de redes de Petri não-autônomas, pois sua dinâmica depende de condições externas e permitem modelagem de aspectos importantes dos sistemas, tais como relações temporais e dependências de sinais externos.

Na Figura 9, resolvemos exercer algum controle sobre o relógio apresentado anteriormente. Para isso, adicionamos à rede da Figura 7 os conceitos IOPT de sinal e evento. Assim, o contador do relógio passará a sofrer influência do ambiente com o sinal `SW1` que representa o acionamento de um switch, que desencadeia o evento de saída `Tick`.

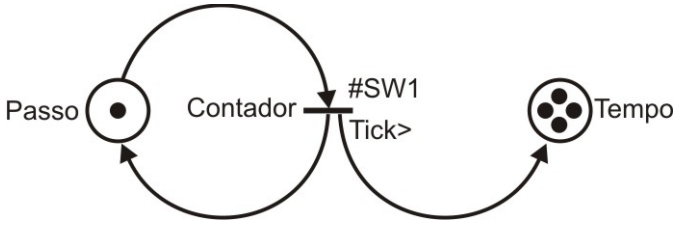

Figura 9: Rede IOPT simples que ilustra fragmentos de um modelo domótico

A Figura 10 apresenta um exemplo mais complexo de um modelo IOPT no contexto de um controlador domótico. Domótica é uma tecnologia recente que permite a gestão de todos os recursos habitacionais. Neste caso, temos um controlador de luminosidade de um determinado ambiente, tal e qual a rede anterior da Figura 8. O modelo reusa a estrutura básica de grafos das redes de Petri. Assim, temos que o lugar `waitingDark` modela a espera por escuridão de um sensor de luminosidade. Ao detectar escuridão, ele passa a marca para o lugar `light`, significando que a luz do ambiente foi ativada. O sistema voltará ao estado de espera através de um comando desligar ou pela detecção de brilho natural pelo sensor. Assim, temos cadastrado nas transições os eventos (OFF, BRIGHT e DARK) e os sinais específicos de alimentação do microcontrolador (SW8, SW9 e SW10). Além do mais, temos sinalizadores, ou sinais de saída, `#LED1` e `#LED2` em cada um dos lugares da rede indicando o estado do sistema. Contudo, as características mais importantes não podem ser observadas sintaticamente, por estarem representadas apenas no domínio semântico. Exemplo disso seria a semântica de execução específica para o modelo, as decisões de prioridade que escolhem qual transição será disparada em uma situação de conflito ou a ocorrência de sinais e eventos que influenciam as guardas das transições das redes de Petri.

Geralmente, as redes não-autônomas conseguem adicionar as seguintes características às redes

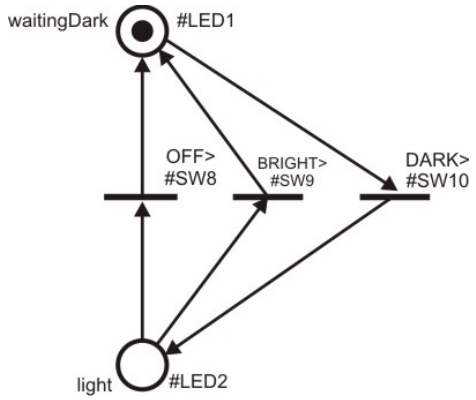

Figura 10: Rede IOPT simples que ilustra fragmentos de um modelo domótico

de Petri: (i) integração com as características gráficas de sistemas de controle, tais como sinais de controle; (ii) capacidade de arcos testarem se lugares possuem alguma marca; e (iii) capacidades para integrar dependências temporais.

As IOPTs reusam as capacidades das redes não-autônomas e ainda se destacam pelas seguintes características de modelagem: (i) prioridades em transições; (ii) atributos de limite para lugares; (iii) um nível de arestas para eventos de entrada; (iv) dois tipos para valores de sinais de entrada e de saída; (v) uma especificação explícita para conjuntos de transições conflitantes; (vi) uma especificação explícita para conjuntos de transições síncronas; e (vii) arcos de testes.

Muitas dessas características se refletem em uma qualidade maior nas transformações que envolvem esses modelos, como por exemplo, a geração automática de código, a resolução de conflitos e o particionamento.

No projeto de sistemas embarcados, IOPTs são usadas para modelar a parte de controle do sistema. O controlador pode ser caracterizado através de dois componentes principais:

1. Descrição da interação física com o sistema controlado ou também chamada de *interface do controlador*;

2. Descrição do modelo comportamental, que é expresso através de um modelo IOPT.

Para a definição da interface do controlador (concebido em [Pais et al., 2005]) em uma rede IOPT é uma tupla $ICS = (IS, IE, OS, OE)$, satisfazendo aos seguintes requisitos:

1. IS é um conjunto finito de sinais de entrada;

2. IE é um conjunto finito de eventos de entrada;

3. OS é um conjunto finito de sinais de saída;

4. OE é um conjunto finito de eventos de saída;

5. $IS \cap IE \cap OS \cap OE = \emptyset$.

A definição das redes IOPT assume o uso de uma linguagem de inscrição, como uma sintaxe concreta, permitindo a especificação de expressões algébricas, variáveis, e funções para a especificação das guardas de transição e condições nas ações de saída associadas aos lugares.

Apresentamos a definição das IOPTs [Pais et al., 2005]. Temos que o conjunto de expressões booleanas é nomeado por BE e a função $Var(E)$ retorna o conjunto de variáveis em uma dada expressão E.

Dado um controlador com uma interface $ICS = (IS, IE, OS, OE)$, uma rede IOPT é uma tupla $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ satisfazendo os seguintes requisitos:

1. P é um conjunto finito de lugares;

2. T é um conjunto finito de transições (disjunto de P).

3. A é um conjunto de arcos, tal que $A \subseteq ((P \times T) \cup (T \times P))$.

4. TA é um conjunto de arcos de teste, tal que $TA \subseteq (P \times T)$.

5. M é a função marcação: $M : P \to N$.

6. $weight : A \to N$.

7. $weightTest : TA \to N$.

8. $priority$ é uma função parcial aplicando transições a inteiros não-negativos: $priority : T \rightharpoonup N$.

9. isg é uma função parcial de guardas de sinal de entrada, aplicando transições a expressões booleanas (aonde todas as variáveis são sinais de entrada): $isg : T \rightharpoonup BE$, onde $\forall eb \in isg(T), Var(eb) \subseteq IS$.

10. ie é uma função parcial de sinal de entrada, aplicando transições a eventos de entrada: $ee : T \rightharpoonup IE$.

11. oe é uma função parcial de evento de saída, aplicando transições a eventos de saída: $ee : T \rightharpoonup OE$.

12. osc é uma função condição de sinal de saída, aplicando de lugares a conjuntos de regras: $osc : P \rightarrow (RULES)^*$, onde $RULES \subseteq (BES \times OS \times N)$, $BES \subseteq BE$ e $\forall e \in BES, Var(e) \subseteq ML$ com ML sendo conjunto de identificadores para cada marcação de lugar após um dado passo de execução: cada marcação de lugar tem um identificador associado, no qual é usado quando o código gerado estiver sendo executado.

As redes IOPT possuem uma semântica *maximal step*: sempre que uma transição estiver habilitada, e a condição externa associada for verdadeira (as guardas dos eventos de entrada e dos sinais de entrada forem verdadeiros), a transição é disparada. O paradigma de sincronização também implica que a evolução da rede é possível apenas em instantes específicos denominados por *tics*. Estes são definidos por um clock global externo. Um passo de execução é o período entre dois *tics*.

Nas definições a seguir, disponíveis em [Costa, 2010], $M(p)$ denota a marcação do lugar p em uma rede com marcação M, e $\bullet t$ denota os lugares de entrada de uma dada transição t ou um dado conjunto de transições S: $\bullet t = \{p|(p,t) \in A\}$; $\bullet S = \{p|(p,t) \in A \wedge t \in S\}$, $\diamond t = \{p|(p,t) \in TA\}$; $\diamond S = \{p|(p,t) \in TA \wedge t \in S\}$.

Definição 2.2 (Condição de habilitação) *Dada uma rede* $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ *e uma interface de sistema* $ICS = (IS, IE, OS, OE)$ *entre* N *e um estado do sistema de entrada* $SIS = (ISB, IEB)$*, uma transição* t*, sem conflitos estruturais, está habilitada a disparar, em um dado* tic*, se e somente se as seguintes condições são satisfeitas:*

1. $\forall p \in \bullet t, M(p) \geq weight(p,t)$.

2. $\forall p \in \diamond t, M(p) \geq weightTest(p,t)$.

3. *A guarda do sinal de entrada da transição* t *avalia para verdade para a seguinte ligação de sinal de entrada:* $isg(t) < ISB >$.

4. $(ie(t), true) \in IEB$.

Definição 2.3 (Passo de uma rede IOPT) *Seja* $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ *uma rede e* $ICS = (IS, IE, OS, OE)$ *uma interface de sistema entre* N *e um sistema com estado de entrada* $SIS = (ISB, IEB)$*. Seja também* $ET \subseteq T$ *o conjunto de todas transições habilitadas em redes de Petri. Então,* Y *é um passo em* N *se e somente se a seguinte condição é satisfeita:*

$$Y \subseteq ET \wedge \forall t_1 \in (ET \backslash Y), \exists SY \subseteq Y, (\bullet t_1 \cap \bullet SY) \neq \emptyset \wedge \quad \exists p \in (\bullet t_1 \cap \bullet SY),$$
$$(weight(p, t_1) + \sum_{t \in SY} weight(p, t) > M(p))$$

Um passo de rede IOPT é máximo (*maximal step*). Isto significa que nenhuma transição adicional pode ser disparada sem estar em conflito efetivo com alguma transição no passo máximo escolhido. Uma ocorrência de um passo em uma rede IOPT e a marcação sucessora respectiva é definida como a seguir:

Definição 2.4 (Ocorrência de um passo e marcação sucessora) *Dada uma rede* $N = (P, T, A, TA, M, weight, weightTest, priority, isg, ie, oe, osc)$ *e uma interface de sistema* $ICS = (IS, IE, OS, OE)$ *entre* N *e um sistema com estado de entrada* $SIS = (ISB, IEB)$*, a ocorrência de um passo* Y *na rede* N *retorna a rede* $N' = (P, T, A, TA, M', weight, weightTest, priority, isg, ie, oe, osc)$*, igual à rede* N *exceto pela marcação sucessora* M' *a qual é dada pela seguinte expressão:*

$$M' = \left\{ \left(p, m - \sum_{t \in Y \wedge (p,t) \in A} weight(p, t) + \sum_{t \in Y \wedge (t,p) \in A} weight(t, p)\right) \in (P \times N_0) \middle| (p, m) \in M \right\}$$

2.3.1 A Transformação Splitting e Modelos Particionados

O modelo apresentado na Figura 10 já possui um grau de refinamento adequado para ser implantado em uma plataforma específica de sistema embarcado com os recursos disponíveis. Contudo, certamente este modelo já foi sub-modelo de algum sistema maior, sendo conseqüentemente resultado de alguma operação de fragmentação. Dentre esses tipos de operações de fragmentação, nos interessamos particularmente pela operação *Splitting*. Esta operação é baseada na definição de um subconjunto de componentes do modelo válido, que permite dividir o modelo original de redes de Petri em diversos sub-modelos para plataformas específicas que se comunicam através de canais síncronos.

A decomposição do modelo é alcançada usando um conjunto de três regras de transformações MDA. Estas regras excluem lugares como nós de cortes, que são parâmetros da transformação,

sempre que o lugar for envolvido em uma situação de conflito. As situações restantes podem ser caracterizadas através de três tipo de regras (regras #1, #2 and #3). A regra #1 é para o caso quando o nó do conjunto de corte é um lugar, e as outras duas regras para o caso quando o nó no conjunto de corte é uma transição.

A regra #1 é ilustrada na Figura 11. Ao escolher remover o elemento `P1` do conjunto de corte da rede inicial (11.a), temos a nova rede fragmentada gerada (11.b) que precisa ter seu comportamento verificado com relação à rede inicial.

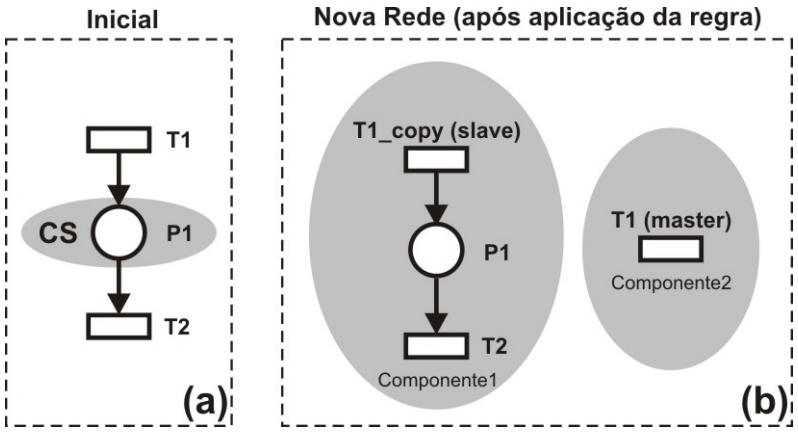

Figura 11: Aplicação da regra #1

A regra #2, ilustrada na Figura 12, é o caso onde o nó do conjunto de corte é uma transição com arcos de entrada de apenas um componente. A partir da rede inicial (12.a), removemos o elemento do conjunto de corte, e (12.b) mostra o resultado do particionamento, mas que também precisa ter seu comportamento verificado com relação à rede inicial.

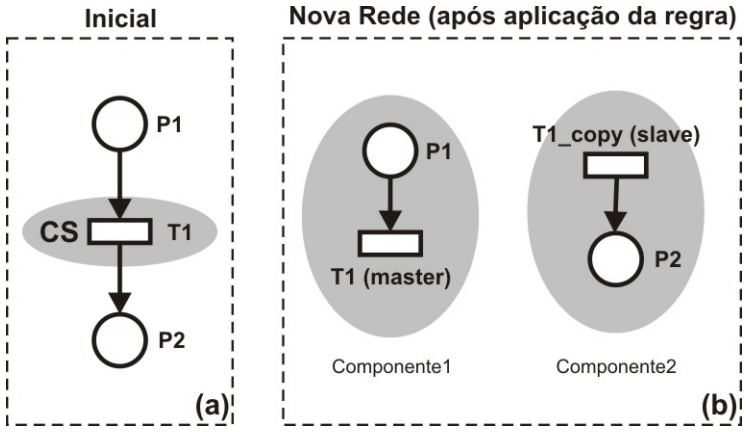

Figura 12: Aplicação da regra #2

A regra #3, ilustrada na Figura 13, é para o caso aonde a transição de corte tem arcos de entrada

de nós que pertencem a sub-redes diferentes após a remoção dos nós. Após o particionamento, um componente irá receber o atributo mestre enquanto os outros irão receber o atributo escravo. A partir da rede inicial (13.a), teremos a operação de remoção de nó, com (13.b) ilustrando o resultado do particionamento, mas que também precisa ter seu comportamento verificado com relação à rede inicial.

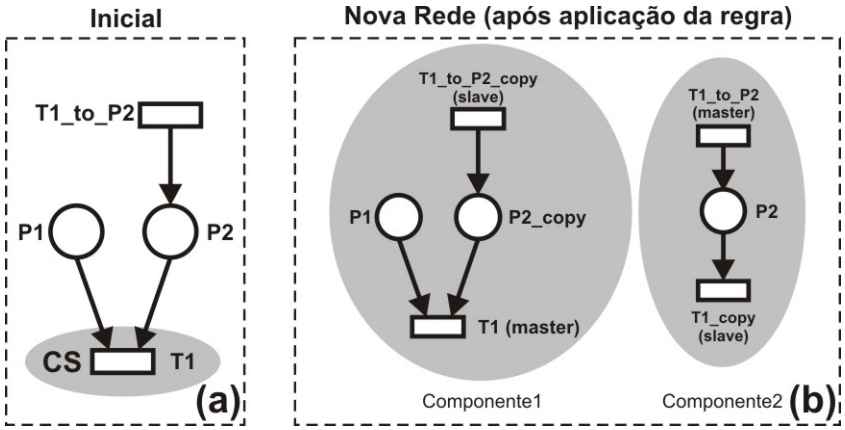

Figura 13: Aplicação da regra #3

Os modelos gerados pela operação Splitting apresentam o conceito de conjunto síncrono (*SynchronySet*), sendo uma rede IOPT com canal síncrono. Uma transição incluída em um *SynchronySet* específico pode ter tanto atributos *master* como *slave*. Em um canal pode haver diversas transições com atributos *slave*, mas apenas uma transição com atributo *master*. Essas transições *slave* só poderão ser disparadas mediante o disparo inicial da transição *master*. O tempo e a coordenação dessas transições slave será dependente do paradigma de comunicação adotado pela plataforma em questão do modelo. De acordo com [Costa, 2010], a definição de uma rede IOPT com *SynchronySet* é dada como a seguir.

Definição de uma rede IOPT com Canal Síncrono

Iniciamos por uma transição rotulada, que é uma transição em redes de Petri t com um atributo nomeado por $t.label$. Um conjunto de transições rotuladas é um conjunto $LTS = T_m \cup T_s$, onde: (i) $T_m = \{t\}$ and $t.label = master$; (ii) T_s é um conjunto de transições rotuladas onde $\forall t' \in T_s$ t'.label = slave e $|T_s| \geq 1$. Temos $t \in T_m, \forall t' \in T_s \rightarrow t \neq t'$. Finalmente, um evento interno é um elemento que será gerado pelo disparo de uma transição com um atributo *master*. Ele contém um atributo $E.master$ que indica sua fonte.

Fazemos a composição dos conceitos apresentados anteriormente em um *SynchronySet*. Ele é uma tupla SS = (ch,LTS,ev), onde:

- ch é um identificador do canal;
- LTS é um conjunto de transições rotuladas;
- ev é um evento interno gerado por $t \in LTS.T_m$.

A condição de ativação de um *SynchronySet* SS é dada se $t \in SS.LTS.T_m$ estiver ativada. Já a semântica de execução de SS é dada se $t \in SS.LTS.T_m$ estiver habilitada e executar, então $\forall t'_j \in SS.LTS.T_s$ ficam habilitadas e executam também logo a seguir. Se o *SynchronySet* incluir um evento interno gerado pela transição *master* e as transições *slave* dispararem em um tempo qualquer após o disparo dessa transição *master*, dizemos que existe um canal de comunicação direto.

Uma rede IOPT com canal síncrono direto é definido como uma tupla (N, S), onde:

- $N = (P, T, A, TA, M, weight,\ weightTest, priority, isg, ie, oe, osc)$ é uma rede IOPT,
- S é um *SynchronySet* válido tal que $S = \bigcup_i SS_i$ onde $SS_i = (ch_i, LTS_i,\ ev_i)$ e $\forall i(t \in SS_i.LTS.T_m \subset N.T$ e $N.oe(t) = N.oe(t) \wedge SS_i.ev(t))$ e $\forall i(\forall t' \in SS_i.LTS.T_s \subset N.T$ e $N.ie(t') = N.ie(t') \wedge SS_i.ev(t))$.

A semântica de execução de uma IOPT com canal síncrono é dada como a seguir. Dado um *SynchronySet* SS, incluso em rede IOPT, o disparo de SS está de acordo com o paradigma de atraso zero, que significa que, considerando t_j uma transição habilitada, $\forall t_j \in SS.LTS.T_s$ disparam ao mesmo passo que $t \in SS.LTS.T_m$. Um passo é composto de dois micro-passos, primeiro disparar $t \in SS.LTS.T_m$ e então disparar $\forall t_j \in SS.LTS.T_s$.

2.3.2 Propriedades dos Modelos

Sobre os modelos construídos nas redes de Petri e redes IOPT, sabe-se bem que a corretude e equivalência de sistemas concorrentes é intrinsecamente mais difícil de analisar do que seqüenciais. Um dos motivos dessa dificuldade é a complexidade envolvida para expressar as interações dos agentes ou componentes do sistema, gerando questões como *deadlock*, *condições de corrida*, *vivacidade*, e outros casos. Além do mais, a maioria dos sistemas concorrentes interagem bastante com seus ambientes, sendo reativos. Assim, esses sistemas geralmente não terminam, ficam eternamente esperando por estímulos do ambiente, o que dificulta verificar corretude através de conjuntos de entradas e saídas. Algumas das principais propriedades comumente detectadas em

sistemas concorrentes são classificadas a seguir. A existência dessa classificação permite nortear estratégias de verificação para algum propósito. Apresentamos como classificá-las de acordo com a categoria de corretude.

- Alcançabilidade: garante-se que um determinado estado satisfatório será alcançado pelo progresso normal do sistema.
- Limitação (boundedness): tipo especial de alcançabilidade que garante que um processo em execução não irá consumir acima de um número limite de recursos.
- Vivacidade (liveness): quando garante-se que o sistema faz sempre progresso e, se desejado, em algum momento futuro, ele irá parar.
- Eqüidade (fairness): garantia de distribuição coerente de recursos entre processos em execução.
- Condição de corrida (race conditions): considerando-se duas seqüências de operações que executam concorrentemente, ela existirá se o resultado das seqüências depender da chegada relativa em algum ponto crítico dessas seqüências.
- Fome (starvation): se alguma das execuções do sistema deixar de progredir devido à presença de outra execuções. Geralmente isso pode ser resolvido por prioridades ou sincronização.
- Deadlock: resulta de uma cadeia cíclica de dependência, sendo observado quando o sistema para ou deixa de apresentar progresso.
- Livelock: é similar ao deadlock, mas ao invés de parar, o sistema entra em um ciclo sem fim de operações.
- Não-determinismo: em caso de escolha de execução ou resultados, qualquer possibilidade poderá ser considerada.

2.3.3 Regras de Equivalência Simples entre Modelos Redes de Petri

Redes de Petri possuem definições básicas de equivalências sintáticas entre modelos. Em [Murata and Koh, 1980], elas são definidas para garantia de equivalência entre algumas propriedades dos modelos. Denominamos essa lista por Π, conforme apresentada na Figura 14,

baseando-se em equivalências que preservam propriedades, como a vivacidade por exemplo. Os padrões são classificados como: (i) Figura 14(a): Fusão de Séries de Lugares (FSP); (ii) Figura 14(b): Fusão de Séries de Transições (FST); (iii) Figura 14(c): Fusão de Lugares Paralelos (FPP); (iv) Figura 14(d): Fusão de Transições Paralelas (FPT); (v) Figura 14(e): Eliminação de Lugares de Auto-Laço (ESP); e (vi) Figura 14(f): Eliminação de Transições de Auto-Laço (EST).

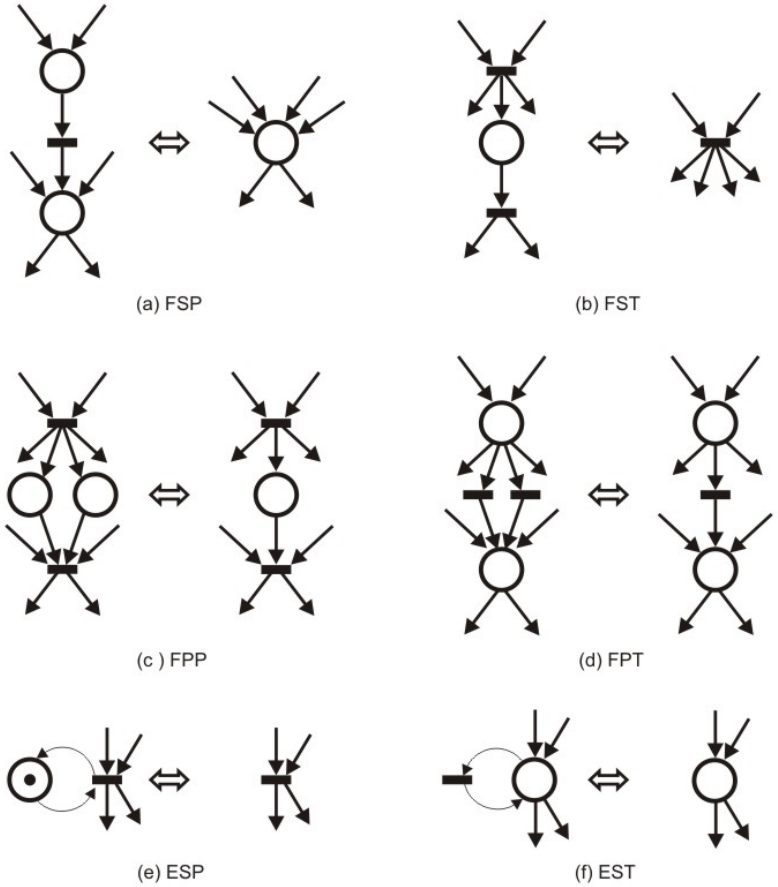

Figura 14: Conjunto de regras de equivalência simples que compõem Π

Para cada uma das regras da transformação Splitting, podemos caracterizá-las de acordo com essa noção de equivalência. Para as regras de transformação 1 e 2, definiu-se uma noção de equivalência, através da lista Δ, ao refinar a equivalência FST (Fusão de Séries de Transições) pertencente à Π da forma como está apresentada na Figura 15. Lá, percebemos que, a partir de um modelo com uma transição em redes de Petri na Figura 15(a), existirá uma ou mais transições (Transition) que serão mapeadas para canais síncronos (SynchronySet) nas redes IOPT, conforme as Figuras 15(b) e 15(c). Este padrão ocorre mesmo que uma diferença sutil ocorra na implementação desses canais, conforme diferenciam-se as redes destas figuras. Na Figura 15(b), tem-se o mapeamento

para uma plataforma assíncrona, onde o conceito da mensagem originada pela comunicação poderá ser representada através da inclusão de um lugar intermediário entre essas transições, enquanto que na Figura 15(c) tem-se o mapeamento para uma plataforma síncrona com consumo instantâneo da mensagem originada pela comunicação.

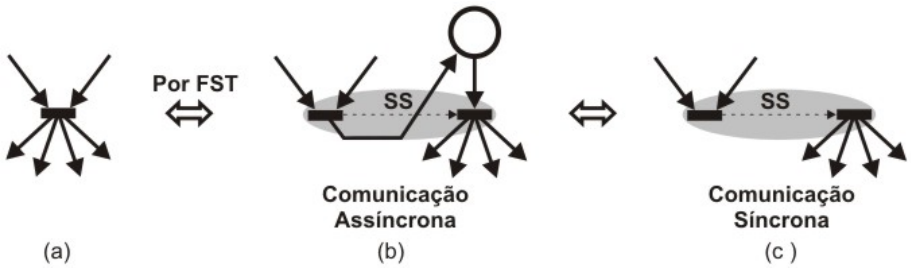

Figura 15: Δ como equivalência simples FST como base para construção das regras #1 e #2

Para a regra #3 da Splitting, especificar Δ é um pouco mais complexa e menos confiável, pois envolveu a composição de duas aplicações da equivalência FST (Fusão de Séries de Transições) com uma da equivalência FPP (Fusão de Transições Paralelas), ambas pertencentes à Π. Isto está representado na Figura 16. Essa especificação estabelece que, através de duas aplicações da regra FST, dois canais síncronos (SynchronySet) devem ser necessariamente criados, guiando a construção de fórmulas de verificação para este cenário, como mostram as Figuras 16(d) e 16(e). A aplicação da regra FPP, assume a abstração da rede na Figura 16(c) como um único lugar, algo comum em algumas técnicas de abstração das redes de Petri, mas que necessitam de verificação de comportamento dos modelos para se ter garantia de preservação de semântica.

2.4 Técnicas Formais

Esta seção descreve as principais técnicas formais envolvidas para construção e validação da solução. Iniciamos com uma descrição dos principais formalismos semânticos envolvidos. Depois recorremos ao uso da lógica de reescrita como um framework semântico unificador para todas as visões apresentadas, estabelecendo uma semântica algébrica translacional. Em seguida apresentamos a técnica de verificação de modelos, que fará a verificação de comportamento para um modelo específico de acordo com propriedades estabelecidas.

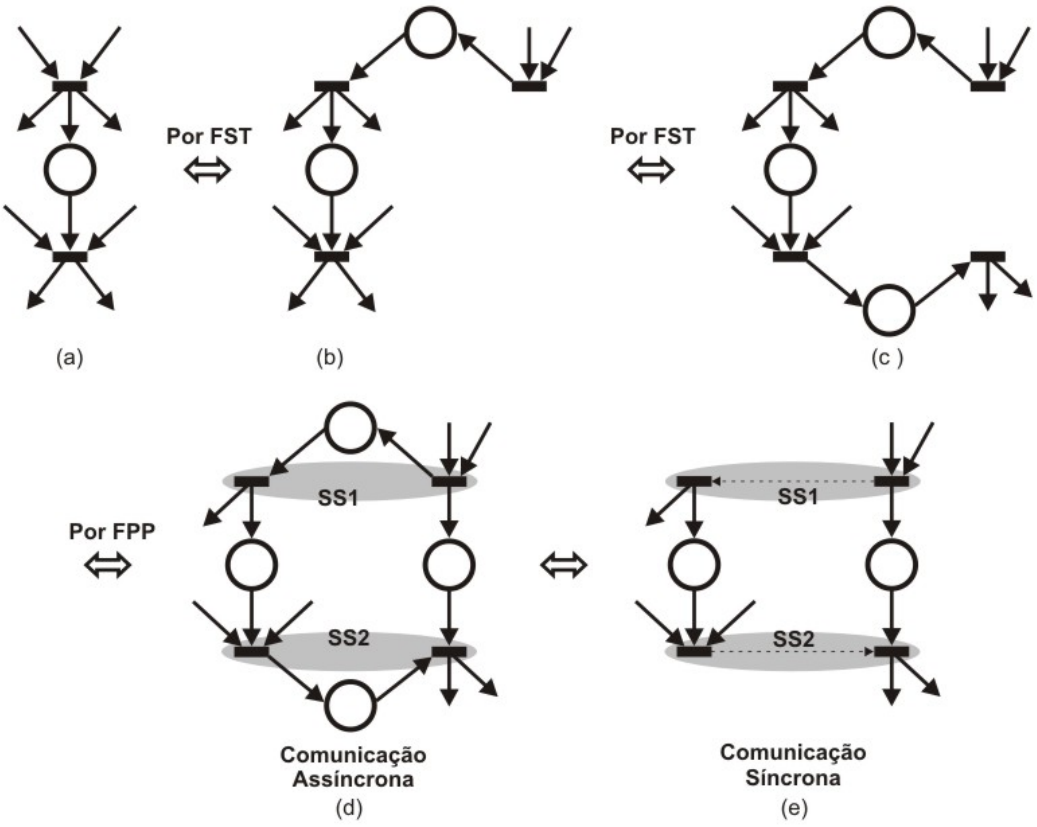

Figura 16: Δ como composição das equivalências simples FST e FPP para construção da regra #3

2.4.1 Conceitos de Semântica Formal

A semântica formal de um modelo é a atribuição de significados para as suas sentenças ou componentes [Tennent, 1976]. Esta atribuição é dada por um modelo matemático que representa toda a computação possível na linguagem que descreve o modelo. Abordagens de semântica formal são requisitadas em vários domínios, também acontecendo na maioria dos tipos de transformações. Elas precisam combinar visões estruturais e comportamentais.

A *semântica estática* de um modelo é a descrição das restrições estruturais de uma linguagem (com aspectos que podem ser sensíveis ao contexto). Um ambiente estático seria uma lista finita de identificadores junto com tipos associados e descritores de estado. Essas restrições estáticas devem ser definidas indutivamente através de um conjunto de regras de derivação orientadas a sintaxe. Essas restrições, se vistas como regras, certamente serão muito mais estritamente direcionadas à sintaxe do que regras de avaliação comuns encontradas nas linguagens, não se preocupando com diversos questionamentos que geralmente emergem naquele domínio, como computações que não terminam, por exemplo. De maneira complementar, o estudo da semântica estática também pode

envolver a verificação dos tipos da linguagem.

A *semântica dinâmica* de um modelo está relacionada com suas regras de execução. Ela especifica como executar programas ou modelos escritos em uma linguagem. Essa descrição pode ser baseada em máquina, descrevendo a execução em termos de um mapeamento da linguagem para uma máquina concreta ou abstrata, ou pode ser baseada em linguagem, descrevendo a execução inteiramente em termos da própria linguagem.

Um cálculo formal para especificação da semântica dinâmica é a semântica denotacional. Ela mapeia construções de linguagens para mapeamentos, nos quais são compostos de acordo com a sintaxe abstrata. Para o caso da necessidade de se utilizar uma máquina formal para a execução da semântica dinâmica, temos a semântica operacional. Vejamos a seguir os principais fundamentos de cada uma das visões.

Semântica Denotacional

Um dos métodos mais promissores para resolução dos problemas decorridos da complexidade da semântica formal é a semântica denotacional [Scott and Strachey, 1971]. Um de seus principais objetivos é prover uma fundamentação matemática apropriada para se argumentar sobre programas e linguagens de programação. Ela foi bastante utilizada na definição das linguagens funcionais. Seu principal conceito é a denotação, que caracteriza-se como uma contribuição de uma parte de um programa para o comportamento como um todo. Denotações em um nível mais genérico podem ser definidas indutivamente, geralmente utilizando-se algum formalismo para especificar como as denotações dos componentes são combinadas. Sua tradução, a partir da sintaxe, é obtida através das *equações semânticas*.

A abordagem genérica para se dar a semântica denotacional de alguma linguagem consiste das seguintes partes:

- A sintaxe da linguagem em questão, de uma forma textual ou gráfica.
- O domínio semântico, que de preferência deve ser um domínio bem conhecido e de fácil entendimento baseado em alguma teoria matemática bem definida.
- Um mapeamento semântico, que é uma definição funcional ou relacional que relaciona cada elemento da sintaxe em um elemento do domínio semântico.

Funções são fundamentais na semântica denotacional. Neste caso, o princípio básico é dar significado para uma linguagem: cada construção sintática deve ser mapeada em uma construção

semântica. O fato é que o conjunto de elementos sintáticos e os domínios semânticos devem ser representados em alguma estrutura e funções matemáticas são usadas como mapeamentos. Espera-se que estas funções apresentem uma compatibilidade desejada para essas estruturas.

Para instanciarmos esta idéia, faz-se necessária a definição de uma coleção adequada de significados para os programas gerados pela linguagem. Isto pode ser obtido com um framework chamado *teoria dos domínios* [Scott and Strachey, 1971], que é o estudo de conjuntos estruturados e suas operações. O conceito fundamental nesta teoria é o *domínio semântico*, que é um conjunto de elementos agrupados por compartilharem alguma propriedade em comum. Estes domínios são acompanhados por um conjunto de *operações*, que lidam com elementos deste domínio.

Um domínio e suas operações constituem uma *álgebra semântica*. A álgebra representa a forma dos elementos do contra-domínio de um mapeamento semântico, chamado de função de valoração, que mapeia elementos sintáticos em elementos semânticos de uma linguagem de programação. Assim como na álgebra universal, ela é o agrupamento de um conjunto com as operações fundamentais naquele conjunto. Este formato provê uma definição precisa da estrutura de um domínio e de como seus elementos são usados pelas funções.

Para um melhor entendimento do formalismo *álgebra semântica*, apresentamos um exemplo a seguir. Escolhemos um sub-conjunto da especificação de algumas construções simples de uma linguagem de programação imperativa no intuito de ilustrar a especificação de domínios comumente empregados na programação convencional. A álgebra semântica está no Código 2.2, extraída de [Schmidt, 1986].

Código 2.2 em Domínios Formais. Exemplo de uma álgebra semântica

```
I. Truth Values
  Domain t ∈ Tr = B
  Operations
   true, false: Tr
   not: Tr → Tr
III. Natural Numbers
  Domain n ∈ Nat = N
  Operations
   zero, one, ...: Nat
   plus: Nat × Nat → Nat
   equals: Nat × Nat → Tr
II. Identifiers
  Domain i ∈ Id = Identifier
IV. Store
  Domain s ∈ Store = Id → Nat
  Operations
    newstore: Store
    newstore = λi.zero
    access: Id → Store → Nat
    access = λiλs.s(i)
    update: Id → Nat → Store → Store
    update = λiλnλs.[i↦n]s
```

A álgebra semântica do Código 2.2 possui quatro domínios: (I) o domínio Truth Values (*valores verdade*), correspondendo ao tipo booleano e tem *true* e *false* como valores verdade e *not* como

operações; (II) o domínio Identifiers (*identificadores*), comumente empregado como endereços específicos de memória, sendo um conjunto sem operações; (III) o domínio Natural Numbers (*números naturais*), com seus elementos incontáveis e as operações básicas *plus* e *equals*; e (IV) o domínio Store (*memória*), que modela o armazenamento básico de um computador como um mapeamento dos identificadores das linguagens para seus valores correspondentes. As operações no domínio Store incluem (i) *newstore* para criar um novo espaço de armazenamento; (ii) *access* para acessar um espaço de armazenamento; e (iii) *update* para colocar um novo valor em um espaço de armazenamento. Isto é a modelagem do conceito clássico de manipulação de memória que sempre esteve presente nas linguagens imperativas.

Seguindo a abordagem tradicional da semântica denotacional, as denotações definidas poderiam ser computadas por indução estrutural. Assim, recorrendo ao exemplo dos números naturais, vejamos como seria sua denotação para expressões aritméticas, considerando `S` como o domínio das *stores* e N como o domínio dos naturais `N`. Assim uma expressão aritmética *A*, seria definida como *A*: Aexp $\rightarrow$ (`S` $\rightarrow$ `N`) por indução estrutural. As relações entre estados e números ficariam da forma como está no Código 2.3.

Código 2.3 em Notação de Funções. Denotações para soma

$$A[[n]] = \lambda s \in S.n$$
$$A[[X]] = \lambda s \in S.s(X)$$
$$A[[a_0 + a_1]] = \lambda s \in S.(A[[a_0]]s + A[[a_1]]s)$$

Temos que um valor natural é uma função de uma store, uma variável acessará um valor para um store e uma soma será uma função das denotações dos dois operandos. Então, para algum estado s, é esperado que aconteça a redução presente no Código 2.4.

Código 2.4 em Notação de Funções. Reduções em soma

$$A[[3 + 5]]s = A[[3]]s + A[[5]]s = 3 + 5 = 8$$

Semântica Operacional

A semântica operacional define o significado de um programa como uma função de transições em uma máquina virtual. A semântica operacional preocupa-se mais em como os programas são executados do que meramente com os resultados das computações. Ela começa a partir de um programa mais um estado inicial e segue computações em busca de novos estados. Por exemplo, ao considerar-se o escopo de linguagens imperativas apresentado na seção anterior, um estado é com-

posto de algumas peças importantes, tais como uma memória (*Store*), como o domínio semântico apresentado previamente e um ambiente (*Environment*), que é uma *tabela de símbolos* que mapeia identificadores para localizações da *Store*.

Colocando em termos lógicos essa definição, temos que, por exemplo, o comando de atribuição poderia ter a seguinte semântica: $\langle E, s\rangle \Rightarrow V \quad / \quad \langle L := E\,,\, s\rangle \longrightarrow (s \uplus (L \mapsto V))$. Esta regra envolve duas partes: antes e depois da barra ($/$) de substituição. A primeira parte indica a premissa, que se a expressão E no estado s se reduzir para o valor V, então, na segunda parte teremos, para uma localização L de memória, que o programa $L := E$ irá atualizar o estado s com a atribuição $L = V$ através do operador de soma a um conjunto representado por $\uplus$.

Para finalizar, ilustramos uma definição da semântica da operação de soma para números naturais. Suas regras de avaliação são apresentadas no Código 2.5. A partir das premissas, temos a atualização de um estado do programa.

Código 2.5 em Regras Operacionais. Avaliação de soma

$$< a_0, s > \rightarrow n_0 \qquad < a_1, s > \rightarrow n_1 \qquad / \qquad < a_0 + a_1, s > \rightarrow n$$

2.4.2 Lógica de Reescrita como um Framework Semântico Unificador

Definições semânticas precisam de alguma plataforma para descrição que satisfaça os requisitos de uma boa representatividade e executabilidade. Em especial, para este trabalho, também se requer uma boa interoperabilidade com os artefatos sintáticos gerados pelos padrões da MDA. Lógica de reescrita pode ser usada como um framework para definições executáveis de linguagens de computação, gerando interpretadores para as linguagens definidas que rodam programas diretamente na semântica da linguagem. Isto também gera definições de linguagens que podem ser usadas para raciocínio formal sobre propriedades de linguagens. Como exemplos, temos verificação de modelos, exploração de espaços de estados para programas, prova de teoremas, corretude parcial via lógica de Hoare e checagem de tipos e de segurança. Em resumo, este framework torna-se adequado para aplicação em várias soluções por combinar três fatos [Meseguer and Rosu, 2007]:

- É um framework lógico, flexível e expressivo que unifica as semânticas denotacional e operacional em uma maneira intuitiva, que evita as respectivas limitações dessas abordagens e permite definições sucintas.

- As definições semânticas podem ser executadas diretamente em uma linguagem como o Maude [Clavel et al., 2011], e então podem tornar-se interpretadores eficientes.

- O reuso de ferramentas formais existentes (por exemplo Java ITP ou Maude Model-Checker [Meseguer and Rosu, 2007]) com capacidades de análise.

Como mencionado anteriormente, semântica formal desencadeia conceitos bastante distintos. Para linguagens determinísticas, ou seja, linguagens com um único fluxo de execução, a abordagem denotacional é a mais adequada. Porém, para sistemas concorrentes, a abordagem operacional apresenta claramente mais propriedades interessantes. A proposta da lógica de reescrita é a junção de diferentes níveis de abstração. Na semântica denotacional, os modelos teóricos são identificados pelas equações semânticas e possuem denotações abstratas únicas. Já na semântica operacional, o detalhe prevalece, dando uma descrição formal passo-a-passo dos mecanismos de avaliação da linguagem, sendo uma abordagem mais sintática. Assim, uma teoria de reescrita é uma tripla (Σ,E,R), provendo uma teoria equacional na qual Σ é uma assinatura de operações e tipos, E é um conjunto de equações e R é um conjunto de regras de reescrita. A semântica denotacional é obtida no caso que $R = \emptyset$, pois temos apenas equações semânticas. De maneira complementar, uma abordagem operacional *pura* é obtida se $E = \emptyset$. Neste caso, temos apenas regras de reescrita como entidades puramente sintáticas.

Em lógica de reescrita, estados da computação são classes de equivalência de E, que são elementos abstratos na álgebra inicial $T_{\Sigma/E}$ e uma reescrita com uma regra em R é entendida como uma transição $[t] \rightarrow [t']$ entre estados abstratos. Naturalmente, a depender do paradigma da linguagem em análise, deve-se tender a uma abordagem mais denotacional ou mais operacional. Por exemplo, dado que uma linguagem imperativa e concorrente deve ter um número de equações x e um número de regras y, uma linguagem puramente funcional teria um número muito superior a x de equações e um número muito inferior a y de regras de computação.

Para este trabalho, escolhemos fazer uso da ferramenta Maude [Clavel et al., 2011]. Isto justifica-se pelo poder de expressividade da sua linguagem, permitindo trabalhar com diversas linguagens descritas pelos metamodelos em MDA, por existir um metamodelo no formato *ecore* [Budinsky et al., 2003] que permite interoperabilidade através de transformações entre modelos e textuais com os diversos padrões existentes para MDA e pela capacidade do seu conjunto de ferramentas de verificação, provas e análise, que dispõem de uma implementação bastante eficiente.

2.4.3 Semântica Algébrica

Objetivamos ter as semânticas apresentadas anteriormente empacotadas como uma *álgebra semântica*, definindo operações algébricas, com axiomas e equações, que simulam o comportamento (operacional) sobre os domínios (denotacional). Esta abordagem, de uma maneira geral, é chamada de *semântica algébrica*. Aqui, estamos interessados em uma definição formal baseada em semântica algébrica para sistemas seguindo uma abordagem translacional. Desta forma, poderemos empregar os trabalhos mais bem sucedidos em semântica formal para a linguagem de modelagem que for empregada no trabalho.

A abordagem da semântica algébrica envolve a especificação algébrica dos dados e das construções das linguagens. Sua idéia básica é definir nomes para os diversos tipos de domínios e operações sobre esses domínios, e a partir daí fazer uso de axiomas algébricos para descrever suas propriedades produzindo *tipos abstratos de dados*.

Algumas denominações são importantes na semântica algébrica. A palavra *tipo* (*sort*) refere-se aos tipos de uma linguagem de programação ou de modelagem, que servem para classificar os dados processados pelos programas ou modelos. Uma especificação algébrica contendo vários *tipos* é composta de duas partes: a *assinatura* (*signature*) e os *axiomas* (*axioms*). Finalmente, uma *assinatura* é um par contendo o nome dos *tipos* e suas *operações*.

A principal razão para se empregar a técnica da semântica algébrica é o seu extenso poder algorítmico e teórico disponível para raciocínio. Isto inclui algoritmos para reescrita de termos, unificação e teoremas para indução e completude de raciocínio com relação às álgebras. Muitos problemas neste domínio tornam-se decidíveis ou semi-decidíveis, o que não acontece para diversos outros formalismos lógicos.

Para ilustrar esse poder da semântica algébrica, consideremos a definição de operações numéricas simples em linguagens imperativas. Consideramos sua representação na linguagem Maude, que foi projetada para semântica algébrica, fazendo com que seus programas sejam teorias equacionais. Os números naturais seriam definidos da seguinte forma, de acordo com a aritmética de Peano, como apresentado no Código 2.6. Após a definição do módulo (linha 1) e do tipo (linha 2), a linha 3 define o elemento 0 (zero) como um termo representando um natural e a linha 4 estabelece a relacão de sucessor, que será um outro termo representando um natural. Para que isso faça sentido, equações e operações são definidas como no Código 2.7.

Código 2.6 em Maude. Definição algébrica dos números naturais

```
1: fmod NAT is
2:   sort Nat .
3:   op 0 : -> Nat .
4:   op s_ : Nat -> Nat .
5: endfm
```

No Código 2.3, após a definição do módulo que reusa o módulo definido anteriormente (linha 1), temos a ilustração da definição do número 1 (um), que será o sucessor do zero (linhas 2 e 3). A linha 4 define a operação + (soma) como binária e com as propriedades de associatividade e comutatividade. Finalmente, as linhas 5 a 7 definem equações sobre essa operação, como a soma com zero e a soma com um sucessor.

Código 2.7 em Maude. Equações e operações para os naturais

```
1: fmod NATOPS is pr NAT .
2:   op 1 : -> Nat .
3:   eq 1 = s 0 .
4:   op _+_ : Nat Nat -> Nat [assoc comm]
5:   vars M N : Nat .
6:   eq M + 0 = M .
7:   eq M + s N = s(M + N) .
8: endfm
```

Os módulos dos Códigos 2.2 e 2.3 podem ser divididos como funcionais. Nesses módulos, as equações são especificadas de três formas diferentes: (i) como regras de simplificação como anteriormente; (ii) sem serem usadas para simplificação através do atributo `nonexec`; ou (iii) como atributos equacionais de operadores específicos como, por exemplo, os atributos `assoc` ou `comm`. Formalmente, dada a teoria (Σ,E,R) e dados dois termos $t \in \Sigma$ e $t' \in \Sigma$, um passo de reescrita através das equações em E é dado por $t \xrightarrow{E} t'$. O fechamento transitivo e reflexivo de $\xrightarrow{E}$ é dado por $*\xrightarrow{E}$. Utiliza-se o comando `red` (reduce) para processar essas equações.

Finalmente, módulo podem possuir regras, tornando-se módulos de sistema. Uma regra tem a forma l: $t \xrightarrow{E} t'$, onde t e t' são termos do mesmo tipo, que podem conter variáveis e l é o rótulo da regra. Estas regras representam transições locais concorrentes em um sistema, com a alteração de um estado s para um estado s' incluindo várias alterações concorrentes. Além do mais, elas também podem conter condições para que sua execução seja permitida. Por exemplo, no Código 2.8 temos a definição na linha 3 da regra uma variável contendo um número natural. O novo estado dessa variável conterá o valor do estado anterior acrescido de 1. Utiliza-se o comando `rew`

(rewrite) para processar estas regras. Maiores detalhes sobre conceitos dessa linguagem podem ser encontrados em [M. Clavel and Meseguer, 2000].

Código 2.8 em Maude. Equações e operações para os naturais

```
1: mod UPDATE is including NATOPS .
2:   var X : Nat .
3:   rl [increment] : X => X + 1 .
4: endm
```

Full Maude

Full Maude é uma linguagem definida em Maude com todas as funcionalidades do Maude mais uma notação para a programação orientada a objetos, visões parametrizadas e outras funcionalidades mais. Essa extensão pode ser considerada como uma experiência com novas características de linguagens de programação. Ela implementa uma interface de usuário para a linguagem estendida, através dos módulos META-LEVEL e LOOP-MODE. Há definições de funcionalidades para análise sintática, módulos de impressão na linguagem e interação de entrada e saída. Tudo isto é redefinido com uma eficiência adequada. Maiores detalhes sobre a linguagem podem ser encontrados em [Durán and Meseguer, 2000].

Neste trabalho, a extensão FULL-Maude é essencial por prover-nos as características essenciais de objetos concorrentes, que são utilizadas na representação de modelos particionados em vários módulos com diversas características de execução e de comunicação. A estrutura dos objetos e suas interações permitem representar diversas infraestruturas de plataformas de execução como módulos e o ambiente no qual esses diversos módulos estejam inseridos, considerando variações comumente encontradas em sistemas reais.

2.4.4 Teoria das Categorias

Definir formalmente o comportamento dos modelos é o primeiro passo para verificar propriedades sobre este. Uma boa alternativa para estruturar *modelos semânticos* no domínio concorrente, utilizando semântica algébrica, é a *teoria das categorias* [Lawvere and Schanuel, 1997]. A teoria das categorias é uma forma abstrata de lidar com estruturas matemáticas e as relações entre elas. [Stehr and Csaba, 2001] argumentam que ela provê uma linguagem abstrata para expressar modelos muito diferentes e permite a tradução de construções e propriedades entre modelos, na qual descrevem uma relação particular entre categorias de estruturas algébricas.

Iniciamos a definição de uma categoria C apresentando as seguintes entidades matemáticas que a compõem:

- Uma classe $ob(C)$ de objetos.
- Uma classe $hom(C)$ de morfismos. Cada morfismo f tem um único objeto origem a e um único objeto destino b. Escrevemos $f{:}a \to b$.

Isto faz com que esse formalismo seja considerado atualmente como um framework geral para a semântica algébrica [Manes, 1975]. O foco é colocado sobre as relações (morfismos) e não nas entidades (objetos). Entidades são descritas apenas de uma maneira abstrata, de acordo com sua interação com outras entidades. Além do mais, é possível definir diversas categorias, de acordo com os tipos de entidades a serem descritas.

Complementando a definição de categorias, ainda temos:

- Uma operação binária $\circ$, que é chamada de composição de morfismos. Dado que temos três objetos a, b e c, temos $hom(a, b) \times hom(b, c) \to hom(a, c)$, sendo representado também por $g \circ f$. Essa composição satisfaz duas propriedades:
  - Se $f{:}a \to b$, $g{:}b \to c$ e $h{:}c \to d$, então $h \circ (g \circ f) = (h \circ g) \circ f$.
  - Para todo objeto x, existe um morfismo $1_x{:}x \to x$ chamado morfismo identidade de x, tal que para todo morfismo $f{:}a \to b$, temos $1_b \circ f = f = f \circ 1_a$.

Um dos resultados mais importantes de colocar o foco nas relações é a construção de diagramas. Desta forma, temos como gerenciar a complexidade das definições mesmo com o incremento acelerado destas. Um exemplo destes diagramas que será de bastante utilidade a este trabalho são os pushouts. Consideremos dois morfismos em expansão com um domínio em comum $f\colon I \to M_{in}$ e $g\colon Z \to M_{out}$. Por definição, o pushout desses dois morfismos f e g consiste de um objeto M_p e dois morfismos $g'\colon M_{in} \to M_p$ e $f'\colon M_{out} \to M_p$ para os quais o diagrama na Figura 17 comuta.

Utilizando-se a teoria dos conjuntos, podemos imaginar M_{in} e M_{out} como conjuntos e sua interseção I também como um conjunto. Tomando a relação de inclusão dada através dos morfismos f e g, temos que $f\colon M_p \to M_{in}$ e $g\colon M_p \to M_{out}$. Assim, teríamos a união também sendo dada através desses morfismos da relação de inclusão, sendo o pushout M_p com $g'\colon M_{in} \to M_p$ e $f'\colon M_{out} \to M_p$ com a mesma comutação do diagrama da Figura 17.

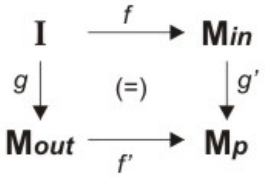

Figura 17: Diagrama Pushout da Teoria das Categorias

Conforme mostrado na operação algébrica de composição e nos diagramas, a teoria das categorias oferece maneiras de combinar entidades. Isto permite que ela seja aplicada com sucesso em situações onde a interoperabilidade seja um ponto crucial. Ela contribui com a semântica algébrica apresentando boas alternativas para representação e execução, além de facilitar sua implementação. Um ganho direto dessa formalização é a modularidade e reuso do framework, pois operações são definidas de uma maneira sucinta e modular, permitindo que os mesmos conceitos sejam usados em outras análises de equivalência de modelos em transformações. Algumas imagens de relações entre modelos apresentadas no trabalho seguem esse diagrama, de acordo com a Categoria Petri que será definida logo a seguir.

A Categoria Petri

A formalização apresentada em [Meseguer and Montanari, 1988] criou a categoria Petri. De acordo com este trabalho, esta categoria é formada por grafos direcionados ordinários equipados com duas operações algébricas correspondendo às composições seqüencial (**;**) e paralela de transições ($+$), representando comportamento, com uma lei distributiva entre essas composições. Este entendimento de concorrência em termos de estruturas algébricas sobre grafos e categorias é uma formalização bastante adequada, pois unifica diversas visões de formalismos semânticos e nos guia na direção de extração de um significado preciso dos modelos.

O grafo ordinário possui um conjunto de nós como um monóide comutativo ($S^{\oplus}$) gerado pelo conjunto (S) de lugares e a marcação vazia como identidade. Para a representação da *semântica dinâmica*, a soma ($+$) de transições representa o disparo paralelo de transições, e o seqüenciador (**;**) de transições representa o disparo seqüencial de transições. Através da garantia da propriedade de fechamento, estamos representando computações através de transições simples. Assim, o trabalho [Bruni et al., 2001] será essencial para extração automática de modelos para esse formalismo. Essa representação algébrica se dá na teoria das categorias pelos motivos apresentados pelo mesmo autor em [Meseguer and Montanari, 1988].

2.4.5 Verificação de Modelos

Verificação de modelos [Edmund M. Clarke et al., 1999] é uma técnica para verificar automaticamente propriedades em modelos formais. Após ter uma descrição comportamental formal do modelo M, propriedades são especificadas para verificar sua veracidade. Essas especificações são geralmente construídas usando lógica temporal proposicional como metalinguagem.

A definição formal desta técnica diz respeito à verificação de uma propriedade. Seja φ um exemplo de uma propriedade, e M um exemplo de modelo, a ferramenta de verificação de modelos executa o processo de verificação algoritmicamente e produz um valor verdade como resultado para indicar se a propriedade φ foi satisfeita ($M \vDash \varphi$), ou não ($M \nvDash \varphi$). No caso da não satisfatibilidade desta propriedade, a ferramenta deve prover uma lista, chamada CE (contra-exemplo) de estados encadeados ($s_0 s_1 s_2 ... s_n$) que foram alcançados no qual demonstram que a especificação não foi válida para esse modelo.

Lógica Temporal Linear e sua Verificação em Lógica de Reescrita

Lógica Temporal é um tipo de lógica modal com modalidades que se referem ao tempo. A linguagem escolhida para especificação de comportamento neste trabalho é LTL (Linear Temporal Logic), logo, as propriedades se restringem ao poder computacional dessa linguagem. Em linguagens de tempo linear, tal como LTL, o tempo é tratado como se em cada momento do tempo, há apenas um futuro distinguível. Assim, as fórmulas são tratadas como seqüencias lineares, descrevendo um comportamento de uma computação simples ou programa. Uma outra possibilidade seria o tempo ramificado, tal como a linguagem CTL, onde cada momento no tempo é dividido em vários futuros possíveis. Neste caso temos árvores infinitas de computação, com cada uma descrevendo o comportamento das computações de um programa não-determinístico. Esta solução está descrita em LTL apenas pelo fato da solução Maude prover o Maude LTL Model-Checker. Esta ferramenta possui uma implementação bastante eficiente e permite a descrição de fórmulas e propriedades de uma maneira bastante intuitiva, combinando a linguagem LTL com os operadores comuns encontrado na linguagem.

A linguagem LTL é construída sobre um conjunto de variáveis proposicionais p1, p2, ..., conectivos lógicos $\rightarrow$, & ($\wedge$), || ($\vee$), ! (~) e operadores temporais:

1. X ou O para próximo

2. G ou [] para sempre

3. F ou <> para futuramente

4. U para até

5. R ou W para entrega

Os modelos utilizados nesse cenário são representados como estruturas de Kripke. Essas estruturas assumem a forma de uma máquina de estados não determinística e representam o comportamento de sistemas. Define-se uma estrutura de Kripke como M = (S, $\underrightarrow{S}$, L), onde S é um conjunto de estados, $\underrightarrow{S}$ é uma relação binária associando dois estados e L é um conjunto de rótulos associando proposições a estados.

Em model-checking LTL, a satisfação de uma determinada fórmula φ em um modelo significa dizer que a estrutura de Kripke associada ao modelo M satisfaz a fórmula em um dado estado s $\in$ S. Assim, temos que $M, s \models \varphi$.

Associa-se uma estrutura de Kripke com uma teoria de reescrita (Σ,E,R) tornando explícito que os estados estarão em Σ e definindo quem são as proposições atômicas relevantes. A ferramenta Maude Model-Checker possui a especificação de satisfação de fórmula, conforme descrita no Código 2.9. A relação de satisfação é definida na linha 4, verificando se dado um estado (`State`) e uma proposição (`Prop`), teremos um resultado de satisfação. Para implementarmos propriedades a serem verificadas por nossos modelos, basta importarmos esse módulo para o módulo onde definiremos nossos predicados e fazermos a declaração de subtipo `subsort MeuTipo < State`.

Código 2.9 em Maude. Construindo predicados a serem verificados pelo Maude LTL Model-checker

```
1: fmod SATISFACTION is
2:  protecting BOOL .
3:  sorts State Prop .
4:  op _|=_ : State Prop -> Bool [frozen] .
5: endfm
```

Para combinar esses operadores temporais em propriedades de mais alto nível, facilitando para usuários não tão familiares com LTL, existem padrões de especificação de propriedades para LTL, conforme definidos por Dill [Musuvathi et al., 2002]. A importância desses padrões é que, além de facilitarem a especificação, eles também podem ser utilizados na geração automática de fórmulas em soluções que sejam fortemente baseadas em verificações de especificações em lógica temporal.

Por exemplo, suponha que P, Q, R sejam propriedades observáveis nos sistemas. Assim, reusaremos neste trabalho diversos desses padrões que especificam propriedades complexas, como por exemplo:

1. Ausência. Assumindo P ser falso, ver Tabela 1.

Propriedade	**Fórmula**
Sempre	[](!P)
Antes de R	<>R -> (!P U R)
Após Q	[](Q -> [](!P))
Entre Q e R	[]((Q and !R and <>R) -> (!P U R))
Após Q até R	[](Q and !R -> (!P W R))

Tabela 1: Padrões para investigação de ausência para uma determinada propriedade

Desta forma, especificamos fórmulas temporais para os seguintes padrões: (i) dado que P é falsa, detecte se ela sempre acontecerá; (ii) dado que P é falsa, detecte se ela ocorrerá antes de uma propriedade R; (iii) dado que P é falsa, detecte se ela ocorrerá após Q; (iv) dado que P é falsa, detecte se ela ocorrerá entre Q e R; e (v) dado que P é falsa, detecte se ela ocorrerá após Q até R. Esses padrões são utilizados em um cenário de verificação de modelos, onde a construção de fórmulas precisa ser manual. Desta forma, eles facilitam o entedimento do que se faz necessário verificar e evita erros de especificação.

2.5 Recapitulação e Considerações

Esta seção recapitula os principais conceitos apresentados nesse capítulo. A Seção 2.1 apresentou as principais características e artefatos definidos segundo a filosofia MDA. A Seção 2.2 apresentou os principais tipos de aplicação de sistemas envolvidos na instanciação da nossa solução. Dada a proposta que temos para transformações preservadoras de semântica, no escopo desse trabalho, foi possível validá-la com aplicação direta para os sistemas embarcados através de projetos colaboradores com esse fim. Nesse cenário, as redes de Petri e IOPTs justificam-se como extremamente relevantes para modelagem nesse contexto e se adequam à filosofia MDA e por isso foram descritas na Seção 2.3. Finalmente, a Seção 2.4 apresentou as principais técnicas envolvidas para a produção desse trabalho. Abordamos os formalismos semânticos empregados para descrição dos

modelos envolvidos nas transformações, o framework de lógica de reescrita para prover a execução necessária à verificação requisitada, a técnica de verificação de modelos que especifica como o comportamento dos modelos envolvidos serão analisados e, para concluir, a equivalência de verificação, que garantirá, a partir dos resultados obtidos no processo de verificação de modelos, a equivalência semântica requisitada pela comparação dos resultados.

3 A Arquitetura MDA-Veritas

A arquitetura MDA-Veritas foi desenvolvida como uma extensão da arquitetura MDA em quatro camadas original. Neste capítulo, descrevemos esta arquitetura generalizando cada um de seus artefatos. Iniciamos com a Seção 3.1, que apresenta a proposta de extensão da arquitetura MDA para incorporar semântica formal em seus artefatos, realizando a concepção de uma técnica para verificação de transformações MDA envolvendo modelos em redes de Petri e suas extensões. Após isso, a Seção 3.2 generaliza um método de instanciação dessa arquitetura, provendo a definição de um processo para verificação de transformações através da MDA-Veritas que pode ser reusada também em outros domínios. Ao final, a Seção 3.3 apresenta conclusões sobre a arquitetura apresentada, apontando materiais complementares existentes.

3.1 A Extensão da Arquitetura MDA de Quatro Camadas

Provemos uma extensão da arquitetura MDA em quatro camadas original, re-apresentada na Figura 18, para lidar com a questão da preservação de semântica em transformações entre modelos. Formalmente, uma transformação entre modelos Tr é vista neste trabalho como uma relação matemática. A partir da relação apresentada em [Troya and Vallecillo, 2010], temos que $Tr \subseteq MM_{in} \times MM_{out}$, onde MM_{in} é um metamodelo de entrada onde é possível existir um modelo de entrada M_{in} instância de MM_{in}, e MM_{out} é um metamodelo de saída onde é possível existir um modelo de saída M_{out} instância de MM_{out}. Desta forma, dizemos que Tr garante, a partir de um algoritmo declarativo ou imperativo, a geração de M_{out} a partir de M_{in}.

Dada uma instância da relação $Tr(M_{in}, M_{out})$, esta será composta por regras, nomeadas como um conjunto R. Assim, cada elemento $r_i \in Tr$ e $r_i \subseteq \wp(M_{in}) \times \wp(M_{out})$, onde $\wp(M_{in})$ representa o conjunto das partes de M_{in} e $\wp(M_{out})$ representa o conjunto das partes de M_{out}. Ambos conjuntos definem padrões nos modelos envolvidos. Assim, na abordagem de transformações empregada por este trabalho, uma instância de regra r_i = ($LeftElements$,$RightElements$) reconhece o padrão $LeftElements$ em M_{in} e produz um novo modelo M_{out} com a substituição do padrão $LeftElements$ pelo padrão $RightElements$. A instanciação e uso da solução que apresentaremos a seguir assume a definição prévia de uma noção de equivalência sintática entre modelos no domínio sendo analisado. Esta noção deve ser extensiva para cada $r_i \in Tr$ e será detalhada nos capítulos posteriores.

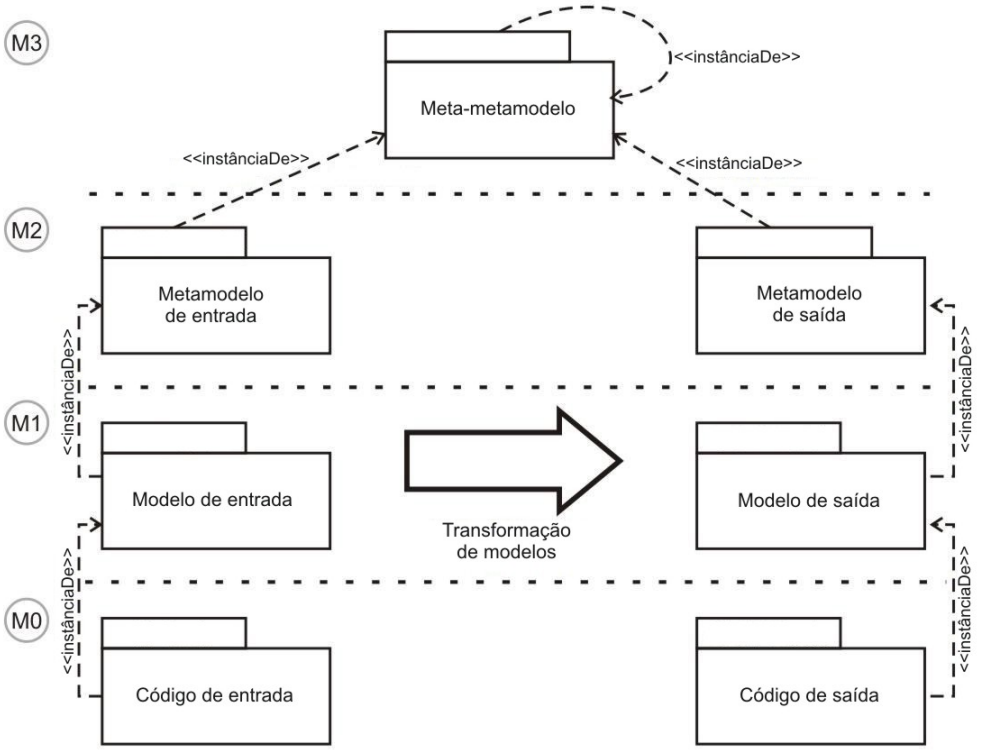

Figura 18: Arquitetura MDA de Quatro Camadas

Tendo o artefato Tr como motor da filosofia MDA, e uma noção de equivalência desejada para as suas regras, a arquitetura MDA-Veritas foi projetada como descrita na Figura 19. Ela estende as camadas M2 e M1 da arquitetura MDA original, para lidar com artefatos envolvidos diretamente com Tr. Para M1 e M2, a MDA-Veritas introduz os seguintes elementos: (i) *Metamodelos Semânticos* MMS_{in} e MMS_{out} em M2; (ii) *Modelos Semânticos* MS_{in} e MS_{out} em M1; (iii) *Equações Semânticas* EqS_{in} e EqS_{out} em M1; (iv) *Tabela de Equivalência Sintática* TES em M1; (v) *Regras de Computação* RC_{in} e RC_{out} em M1 e (vi) *Verificador Formal* $\cong$ em M1. Com o objetivo de permitir a representação da parte estrutural e comportamental da semântica dos modelos, ambas camadas incorporam as visões algébrica, denotacional e operacional da semântica formal através de artefatos representados na cor cinza na Figura 19. Desta forma, RC_{in} e RC_{out} não aparecem nesta visão macro da arquitetura porque são pertencentes ao módulo *Semântica Dinâmica*. Dizemos que a solução é capaz de verificar uma relação de equivalência das definições dadas pelos *metamodelos semânticos*, que são os *modelos semânticos*, através de um *verificador formal* fazendo uso de uma *tabela de equivalência sintática*. Assim, a TES também não aparece nessa visão macro da arquitetura por estar inclusão no módulo $\cong$. Este mecanismo de verificação permite analisar a es-

trutura e o comportamento dos modelos, de uma forma unificadora, como uma *álgebra semântica*, obtida através *equações semânticas* reusando a teoria da semântica denotacional e a computação de estados, como *regras de computação*, reusando a teoria da semântica operacional.

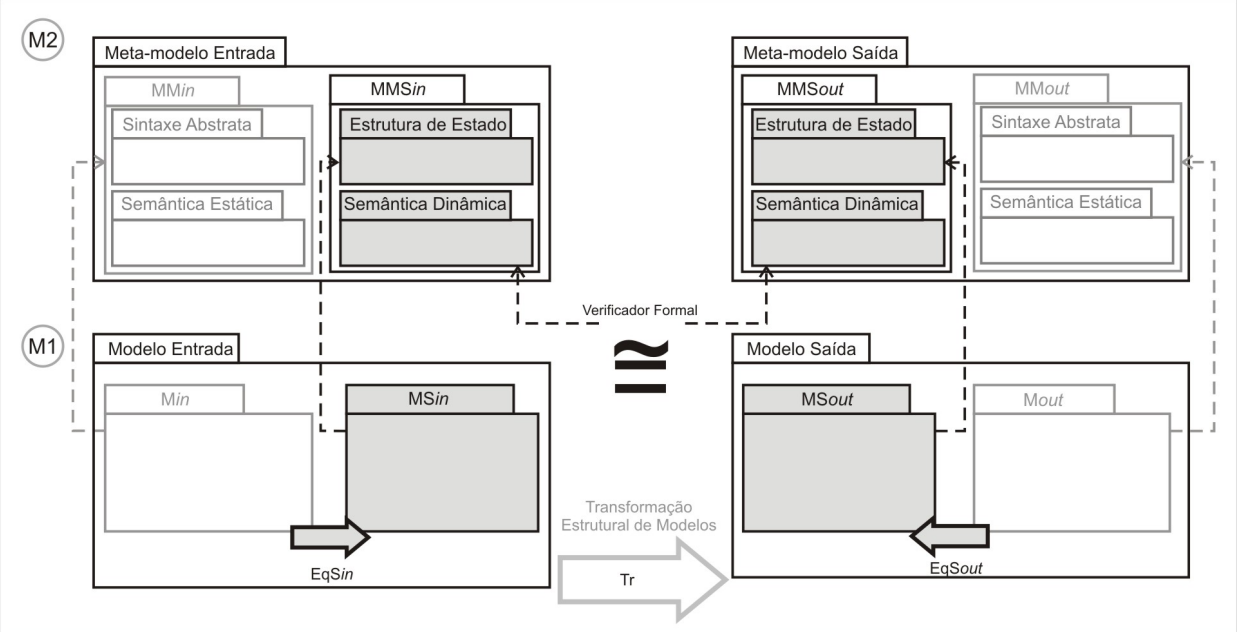

Figura 19: Camadas M1 e M2 na arquitetura MDA-Veritas

O módulo da *Semântica Estática* é definido por MM_{in} e MM_{out}, onde propomos uma separação completa do conceito da sintaxe abstrata deste conceito. Este módulo provê a definição e representação correta dos conceitos envolvidos dependendo de um contexto estático, complementando a definição da linguagem. A implementação de um ambiente para semântica estática checa a qualidade dos modelos. Ela tradicionalmente descreve atividades em tempo de compilação, tais como checagem de tipos, cálculo de armazenamento, resolução de escopo e definições completas e sem ambigüidades de todas variáveis dentro de um bloco de programa ou modelo. Como vários metamodelos que descrevem a sintaxe abstrata das linguagens ainda estão, de fato, incompletos na comunidade MDA [Atlantic Zoo, 2011], este bloco faz-se necessário como um auxílio a evitar as questões acima mencionadas, sempre que aparecerem. Nosso objetivo específico aqui é completar MM_{in} e MM_{out} com restrições que viabilizem o tratamento formal da arquitetura como um todo. Por analogia, este módulo está posicionado em MM_{in} e MM_{out} porque suas relações são geralmente definidas indutivamente sobre um conjunto de regras de derivação dirigidas por sintaxe, não requisitando a aplicação dos artefatos *Equações Semânticas*, que serão definidas logo adiante, aos modelos.

Os *Metamodelos Semânticos* (MMS_{in} e MMS_{out}) dividem-se em dois submódulos: *Estrutura de Estado* e *Semântica Dinâmica*. A *Estrutura de Estado* define a sintaxe abstrata das especificações semânticas, dadas por domínios semânticos compostos por funções matemáticas. Estes domínios exercem um papel de linguagem semântica, que é a linguagem para dar as especificações semânticas propostas e definir a parte estrutural da semântica dos modelos. A *Semântica Dinâmica* tem a responsabilidade de definir uma infraestrutura de regras para modelar o comportamento, usando ou alterando um estado existente. A existência desse metamodelo justifica-se sob vários motivos. Enumeramos estes motivos como a seguir:

1. Provêem interoperabilidade entre formalizações: os *metamodelos semânticos* representam formalizações de teorias bem consolidadas para representação de estrutura e comportamento de modelos. Através de metamodelos dessas formalizações, podemos definir transformações que representem equivalências desejadas entre conceitos, facilitando a tarefa de verificação de preservação de semântica.

2. Provêem interoperabilidade entre ferramentas: diversas soluções MDA provêem gerencia de modelos e aplicação de transformações. Com uma representação da teoria semântica envolvida através de um metamodelo, essas ferramentas poderiam fazer uso desses conceitos como artefatos comuns de MDA, facilitando o ingresso dessas teorias na proposta já existente da MDA.

3. Facilitam geração de código para soluções específicas: há diversas soluções que adotam técnicas formais que são de interesse direto à questão da preservação de semântica. Entre eles, podemos mencionar model-checkers, provadores de teoremas, simuladores e outros mais. Essas ferramentas possuem naturalmente suas linguagens de entrada. Com a existência de *metamodelos semânticos*, essas formalizações podem ser mapeadas diretamente para a sintaxe textual dessas ferramentas, através de transformações modelo-para-texto, facilitando a geração de especificações formais e sua respectiva executabilidade.

Os *Modelos Semânticos* (MS_{in} e MS_{out}) são módulos na camada M1 que instanciam MMS_{in} e MMS_{out} e cujo seu comportamento é descrito pelo módulo *Semântica Dinâmica*. Este artefato representa o resultado da extração dos domínios semânticos de um programa ou modelo, permitindo representar as suas construções e seu comportamento de uma maneira independente da sintaxe.

Equações Semânticas (EqS_{in} e EqS_{out}) são representadas pelas setas pequenas na Figura 19, partindo M_{in} e M_{out} para MS_{in} e MS_{out} na camada M1. Elas definem mapeamentos, baseados em metamodelos, das estruturas da sintaxe abstrata da linguagem para significados desenhados a partir dos domínios semânticos. Esse mapeamento é especificado através de linguagens que sejam compatíveis com os padrões de MDA. Transformações entre modelos podem ser entendidas como equações semânticas usualmente definidas quando se especifica a semântica formal de uma linguagem de programação ou de modelagem.

Para se dar uma semântica formal a um modelo, faz-se necessária a atribuição de significado às suas sentenças ou componentes. Este significado deve ser dado por um modelo matemático que represente toda computação possível na linguagem que descreve o modelo. O significado, dado por EqS_{in} e EqS_{out}, é obtido pelo emprego da semântica denotacional através de funções que mapeiam elementos sintáticos para conjuntos matemáticos bem-definidos. Tecnicamente, a realização desse mapeamento era considerado como um problema em aberto [Engels et al., 2000], recebendo algumas propostas de soluções para paradigmas específicos mas que continua deficiente no paradigma concorrente. Ainda não havia sido dada uma definição formal das *equações semânticas* em trabalhos relacionados à semântica em MDA, e como resultado, os modelos definidos na MDA não significam nada além de árvores de derivação sintáticas. De acordo com [Mosses, 1975], o nível de formalidade das *equações semânticas* para a maioria dos programas não é o bastante para garantir definições completamente precisas e sem ambigüidades. Nossa proposta é resolver esse problema de uma forma pragmática: reusando linguagens e padrões da MDA.

As *Regras de Computação* (RC_{in} e RC_{out}) são representadas como setas no centro da camada M1 e são aplicadas a MS_{in} e MS_{out}. Se a linguagem tiver uma teoria equacional que a descreva, neste caso, a semântica dinâmica poderá ser detalhada como a inferência e configuração de estados de modelos. Assim, arcos representam reduções na direção de uma simplificação do programa original até valores finais computados e armazenados em estruturas auxiliares, tais como memória, ambientes ou tabelas de símbolos utilizando-se diversos tipos de especificações, como por exemplo, as equacionais da semântica algébrica. RC_{in} e RC_{out} também provêem o controle para estados no estilo operacional, permitindo a mecanização de processamento de estados para a inferência de equivalência entre os modelos analisados envolvidos na transformação. É necessária a formalização e escrita dessas regras de acordo com o paradigma da linguagem dos modelos analisados.

Caracterizamos RC_{in} e RC_{out} como qualquer ação de mudança no formato ou observação de

comportamento de MS_{in} e MS_{out} no intuito de inferir propriedades semânticas com um maior grau de precisão. Por exemplo, com relação a dois modelos envolvidos em uma transformação, podemos reduzi-los a uma forma mais simples para que eles possam ser comparados entre si de uma maneira mais trivial, reescrevê-los para que também alcancem uma forma mais próxima da similar e seja permitida essa comparação, ou ainda, aplicar diversas especificações de comportamento, observar como esses modelos se comportam com essas especificações e a partir daí, inferir a equivalência semântica necessária. A escolha do melhor método para essa situação está condicionada ao formalismo semântico dependente do paradigma dos modelos sob análise e da técnica a ser empregada no *verificador formal*.

A *Tabela de Equivalência Sintática* (TES) é o resultado da aplicação de um método de especificação da noção de equivalência da transformação sob análise e tem o propósito de guiar a geração de um conjunto de proposições e fórmulas para verificação de equivalência semântica entre os modelos transformados.

O *Verificador Formal* ($\cong$) é um mecanismo que faz uso da TES e de RC_{in} e RC_{out}, e é requisitado para provar invariantes e propriedades sobre a conformidade de M_{in} e M_{out}. A instanciação desse módulo vai depender do paradigma de M_{in} e M_{out}, e geralmente deve fazer uso de ferramentas bem consolidadas de métodos formais, tais como provadores de teoremas, verificadores de modelos e outras mais.

A definição da arquitetura MDA-Veritas conclui-se com a escolha de uma abordagem baseada na *verificação de equivalência entre modelos*, definindo o tipo de significado para se trabalhar com MS_{in} e MS_{out} ao usar $\cong$. Consideramos o comportamento explícito em si através da estrutura de espaço de estados do modelo ou também possivelmente como uma visão fechada, através das entradas e das saídas, onde as funcionalidades apresentadas ou um conjunto de propriedades devem ser satisfeitas. Além do mais, foi definido que esse tratamento deva ser realizado de uma maneira independente, tanto do processo de aplicação da transformação, quanto nas demais partes a serem produzidas por nossa solução. Assim, a relação de equivalência definida na TES deve ser especificada por algum *especialista*, a partir de M_{in} e M_{out} e de Tr. A relação deve ser definida de acordo com critérios do projeto de quem esteja provendo Tr, que exerce o papel de *cliente* da arquitetura MDA-Veritas.

Os benefícios apresentados pelo método escolhido de extensão da arquitetura são: (i) através de metamodelos, provemos uma sintaxe abstrata para aplicar *álgebra semântica* como uma linguagem

específica de domínio para representação de significado de modelos, sendo complementadas com conceitos correspondentes aos paradigmas dos modelos; (ii) através de transformações, provemos interoperabilidade entre especificações semânticas de diversas linguagens, permitindo processamento de suas denotações; (iii) através de mapeamentos para ferramentas formais, provemos refinamento e computação formal dos modelos semânticos, que podem ser aplicados para analisar vários tipos de transformações [Czarnecki and Helsen, 2003]; e (iv) a arquitetura como um todo apresenta artefatos e construções baseadas nas visões denotacional, operacional e algébrica da semântica formal, permitindo reuso das vantagens de cada um desses métodos.

3.2 Um Processo para Verificação de Transformações MDA de Sistemas Concorrentes

Para que a MDA-Veritas possa ser útil para verificação de transformações de modelos, como esperado, é necessário instanciá-la com técnicas e ferramentas específicas de acordo com o paradigma dos modelos envolvidos. Esse processo é complementar à apresentação dos itens da arquitetura, pois, após a definição em termos abstratos dos artefatos semânticos da MDA-Veritas, o foco se torna suas concretizações. A Figura 20 é um diagrama de atividades UML que representa o fluxo de trabalho desse processo. Cada raia deve conter as atividades que serão específicas para os atores existentes: o *Provedor da Solução* e o *Cliente MDA*. O *Provedor da Solução* é o especialista mencionado anteriormente na discussão sobre o *Verificador Formal*. Este ator deve possuir conhecimento em métodos formais e será o responsável por prover uma solução completa para o *Cliente MDA*, escondendo conceitos formais e aspectos complexos das técnicas e ferramentas.

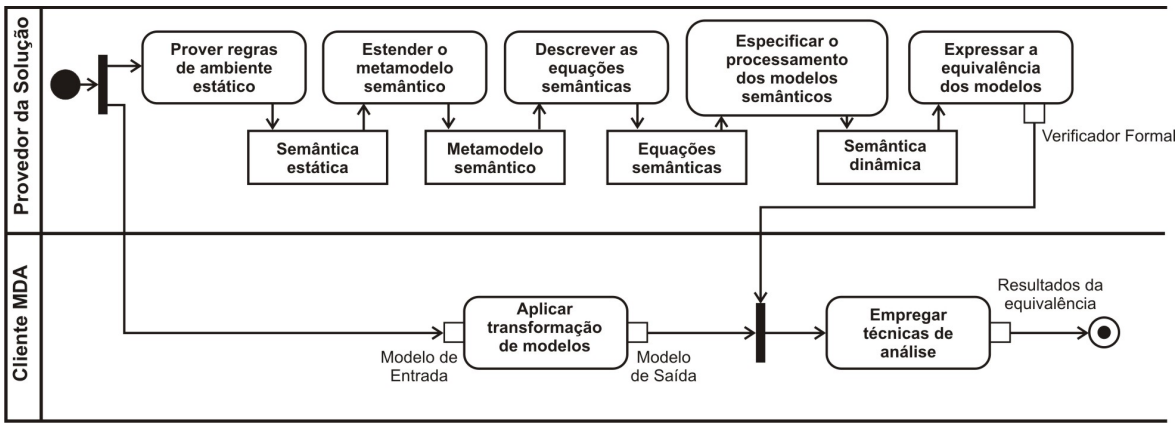

Figura 20: Processo de instanciação esperado

A idéia da MDA-Veritas é geral em princípio para qualquer transformação entre modelos. Contudo, espera-se que o *Provedor da Solução* possa preencher a MDA-Veritas com técnicas específicas dependente do paradigma dos modelos envolvidos na transformação e de sua preferência de uso. Isto está melhor detalhado na parte mais alta da Figura 20, na raia do *Provedor da Solução*. As atividades dessa raia devem ser executadas manualmente, no intuito de que a infraestrutura da solução seja realizada. Assumimos uma definição de noções de equivalências sintáticas entre modelos para cada regra $r_i \in Tr$. A partir daí, a primeira atividade é prover regras de boa formação em um ambiente estático, ou simplesmente regras de boa formação, a partir da sintaxe de M_{in} e M_{out}. Esta atividade pode produzir a *semântica estática* para os modelos a serem analisados a depender da complexidade dessas regras de checagem. Essas regras garantem que devemos lidar apenas com modelos construídos corretamente em nossa solução. A seguir, com o objetivo de prover MS_{in} e MS_{out}, MMS_{in} e MMS_{out} devem ser preenchidos com uma teoria específica de acordo com o paradigma dos respectivos modelos analisados. Após isso, a descrição de EqS_{in} e EqS_{out} faz-se necessária para a extração automática de MS_{in} e MS_{out}. O processamento, ou simplificação, de MS_{in} e MS_{out} é a próxima atividade, produzindo RC_{in} e RC_{out}, que são responsáveis pela *semântica dinâmica* dos modelos. As duas atividades finais vão na direção da obtenção da equivalência desejada entre os modelos. Especifica-se TES a partir do enquadramento da transformação à formalização dos modelos envolvidos. Para cada regra, tem-se que especificar uma noção de equivalência sintática. Finalmente, o *Provedor da Solução* deve escolher uma abordagem de comparação para verificar a equivalência entre os *modelos semânticos*. Ele deve considerar a TES construída e decompor as regras que definem a transformação sob análise a partir dessas equivalências simples. A conclusão desta atividade produz $\cong$ para Tr fazendo uso de ferramentas de verificação formal. O produto de todas essas atividades deverá ser uma instância da arquitetura MDA-Veritas para o paradigma dos modelos envolvidos na transformação.

Ainda sobre a Figura 20, na raia do *Cliente MDA*, temos atividades que são executadas automaticamente. Inicialmente, aplica-se Tr à M_{in}. Então, M_{out} é capturado, no qual, juntamente com M_{in} devem ser submetidos ao produto originado da última atividade de instancição do protótipo para empregar técnicas de análise de equivalência entre esses modelos. Assim, dependemos das regras construídas para Tr, que originarão, automaticamente ou através de especificação manual, fórmulas de especificação a serem verificadas para garantir que propriedades devam ser preservadas nos modelos. O resultado da equivalência permitirá detectar se a transformação é preser-

vadora de semântica para aquele caso ou não, juntamente com alguma interpretação provida pelo módulo $\cong$, como um contra-exemplo, se for o caso.

3.3 Recapitulação e Considerações

Este capítulo introduziu a arquitetura MDA-Veritas, que é o principal pilar deste trabalho e sugeriu um processo para instanciação dos elementos que compõem esta solução. A partir daí, esperamos que os conceitos apresentados possam ser instanciados para diferentes paradigmas, de acordo com os modelos envolvidos nas transformações a serem verificadas. Para maiores detalhes sobre estes processos e artefatos, o leitor deve consultar alguns trabalhos que estão disponíveis em [MDA-VERITAS, 2011]. Por exemplo, a construção de um metamodelo a partir da idéia de *álgebras semânticas* é realizada em [Barbosa et al., 2008b], uma instanciação mais detalhada da MDA-Veritas em linguagens imperativas é discutida em [Barbosa et al., 2008a] e a inserção das equações semânticas na MDA-Veritas foi introduzida em [Barbosa et al., 2010b].

4 Aplicação da MDA-Veritas para Sistemas em Redes de Petri

Este capítulo descreve uma aplicação, através do procedimento de instanciação da arquitetura MDA-Veritas, para que seja possível lidar com a verificação de transformações MDA (Tr) envolvendo modelos (M_{in} e M_{out}) no paradigma concorrente descritos em redes de Petri e redes IOPT. Descrevemos em detalhes como cada elemento da MDA-Veritas pôde ser instanciado, com técnicas, linguagens e artefatos, no contexto de aplicações para sistemas embarcados utilizando modelos em redes de Petri e sua extensão, as redes IOPT.

Provemos uma representação formal de M_{in} e M_{out}, seguindo nossa abordagem para propósitos de verificação de equivalência. Objetivamos, com isso, aplicar nosso trabalho através de uma instância de Tr que representa a idéia central de um processo, que cobre desde a especificação de sistemas, até a implementação proposta pelo *cliente MDA*. Iniciamos com a Seção 4.1, onde efetuamos um planejamento com a escolha das melhores técnicas e soluções para o estudo de aplicação trabalhado. Na Seção 4.2, apresentamos uma representação formal de MS_{in} e MS_{out} para redes de Petri que posteriormente representarão PIMs, e na Seção 4.3, complementamos essa representação formal com modelos IOPT, onde definimos MS_{in} e MS_{out} para esta linguagem, que posteriormente representarão PSMs. Por fim, a Seção 4.4 faz considerações complementares sobre esta instanciação.

4.1 Planejamento do Estudo de Aplicação

Esta instanciação da MDA-Veritas lida com o artefato Tr, explorando uma caracterização formal de suas regras, porém ignorando aspectos específicos de sua implementação. Desde já, o foco se dá em MS_{in} e MS_{out}, em noções de equivalência λ fornecida à solução, nas TES extraída e no método para verificar se a semântica que o *cliente MDA* esteja interessado em analisar é preservada através das transformações. Isto permite aplicar ao modelo transformado (M_{out}) os mesmos resultados de análise do modelo de entrada (M_{in}), com a geração e especificação de um conjunto de propriedades P, no intuito de que sejam preservadas ao aplicarmos $\cong$.

A Figura 21 provê informação sobre os principais formalismos e ferramentas escolhidas que combinam-se no intuito de prover a executabilidade requisitada para nossa abordagem. Esta instanciação está apta a trabalhar com a verificação de equivalência entre M_{in} e M_{out} de qualquer Tr envolvendo redes de Petri.

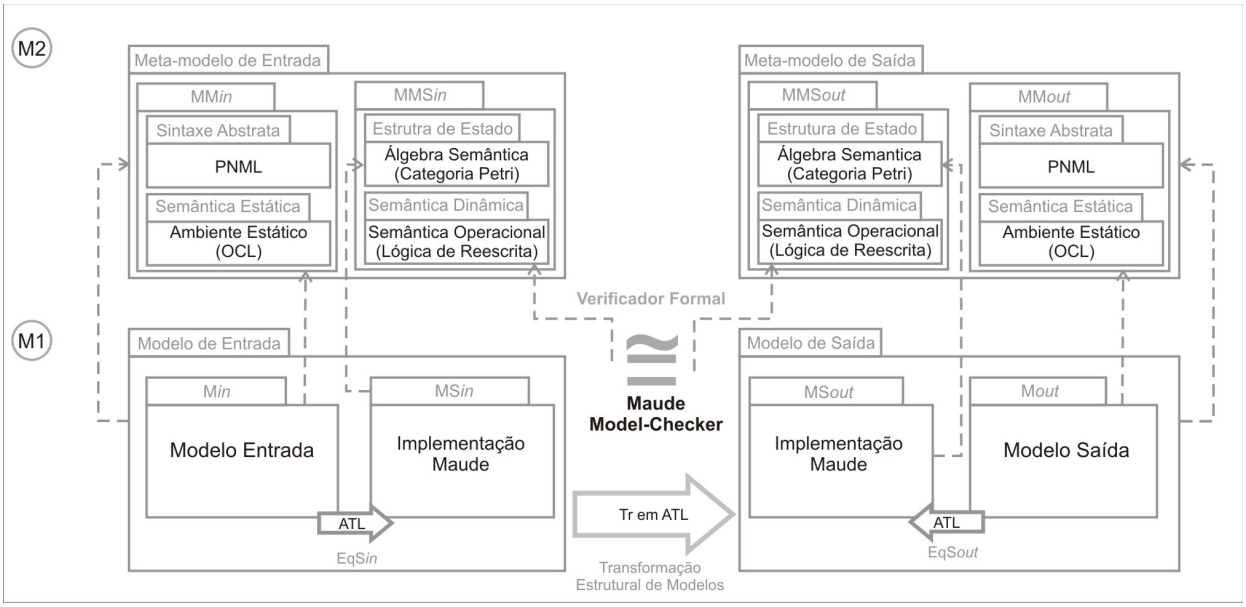

Figura 21: Instanciação da arquitetura para verificação de modelos de redes de Petri

Ainda sobre a Figura 21, sumarizando o que será instanciado durante o restante desse capítulo, temos a extração de MS_{in} e MS_{out} para M_{in} e M_{out} a partir de Tr, que exerce o papel de uma transformação puramente estrutural de modelos. MS_{in} e MS_{out} estarão descritos em uma *categoria* chamada Petri, que recebeu uma implementação no conjunto de ferramentas Maude. Esta representação semântica estará de acordo com MMS_{in} e MMS_{out} que provêem uma *álgebra semântica* específica de acordo com a categoria Petri. Além do mais, o mecanismo utilizado por RC_{in} e RC_{out} pode seguir a semântica operacional da ferramenta Maude, ou é redefinido por reflexão. Finalmente, o mecanismo de $\cong$ fará uso da extensão para verificação de modelos (model-checking) provida também pelo conjunto de ferramentas Maude.

4.1.1 Preâmbulo: Definição de Mapeamentos Sintáticos entre Redes de Petri e IOPTs

Dada Tr, um passo essencial, pré-requisito antes instanciação da solução MDA-Veritas, consiste em estabelecer uma noção de equivalência sintática básica entre modelos para orientar a definição das proposições atômicas e fórmulas a serem empregadas pelo módulo *Verificador Formal*. Esta é uma noção que já precisa existir para os atores envolvidos com a solução para que possam guiar a solução na busca pela verificação de equivalência.

Para cada regra r_i de Tr, definimos uma função parcial λ_ri de rótulos que estabeleça um mapeamento, com assinatura λ_ri: $mc_{in} \rightharpoonup mc_{out}$, onde mc_{in} é uma metaclasse de MM_{in}, ou seja,

$mc_{in} \in MM_{in}$ e mc_{out} é uma metaclasse de MM_{out}, ou seja, $mc_{out} \in MM_{out}$. Formamos assim o conjunto Λ de mapeamentos sintáticos. Assim, o corpo desse mapeamento λ_ri estabelece uma equivalência entre os tipos presentes fortemente baseada na lista Π, através de cada Δ_i definido para cada regra, como apresentada na Seção 2.3.3, para Tr. A função λ_ri é parcial devido à exigência de não se garantir que todos os elementos pertencentes ao seu domínio, instâncias de mc_{in}, sejam necessariamente mapeados à sua imagem, flexibilizando a especificação sintática estabelecida pelo cliente MDA. Desta forma, o cliente MDA pode estar interessado que apenas alguns elementos de M_{in} tenham sua correspondência sintática com M_{out}.

Para o cenário que trabalharemos ao longo deste capítulo, envolvendo redes de Petri e redes IOPT, especificamos o mapeamento como λ_ri: Transition $\rightharpoonup$ Transition ou λ_ri: Transition $\rightharpoonup$ SynchronySet, com i=1 ou i=2 ou i=3, onde Transition é uma metaclasse mc_{in} ou mc_{out} do metamodelo redes de Petri (MM_{in} ou MM_{out}) e SynchronySet é uma metaclasse mc_{out} do metamodelo IOPT (MM_{out}). Para o primeiro caso, que representa um mapeamento entre redes de Petri envolvendo PIMs, dado um elemento $t \in M_{in}$.T, retorne, se existir, $ts \in M_{out}$.T | (t.name + ";"+ t.name + "_copy“) = (ts.name). Já para o segundo caso, que representa um mapeamento entre redes de Petri e redes IOPT envolvendo PSMs, dado um elemento $t \in M_{in}$.T, retorne, se existir, $ss \in M_{out}$.SS | (t.name + "master") = (ss.master.name). Uma boa ilustração desse mapeamento pode ser analisado para a regra #2 na Figura 22. Observe que os conceitos semânticos que definem a regra de transformação foram esquecidos, focando apenas nos conceitos que envolvem a equivalência λ_r2 de acordo com o seu Δ_2. Esta especificação só foi possível porque baseia-se somente nas informações produzidas na modelagem de cada regra $r_i \in Tr$ através das equivalências estabelecidas pelo cliente MDA para a Splitting.

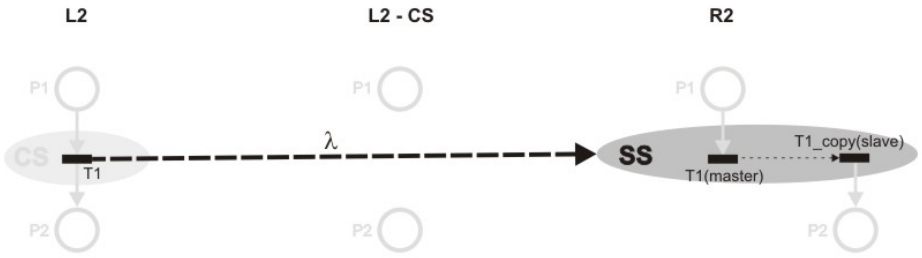

Figura 22: Mapeamento λ_r2 sobre os conceitos que definem a regra #2

Os Códigos 4.1 e 4.2 representam uma implementação de λ_r2 como *helpers* ATL. Em ambos os casos, em sua assinatura (linha 1), recebe-se como parâmetro um elemento `t` instância da

metaclasse `PetriNet!Transition`, porém, para o Código 4.1 retorna-se uma instância da metaclasse `PetriNet!Transition` e para o Código 4.2 retorna-se uma instância da metaclasse `IOPT!SynchronySet`. No corpo das funções (linhas 2-3), seleciona-se entre todas as instâncias da metaclasse de retorno a primeira cujo nome do componente faça o casamento com o componente case em nome com o da entrada. Estes helpers serão utilizados posteriormente pela função de extração da TES, definida nas Seções 4.3.2 e 4.2.5, que será posteriormente gerada como uma pequena transformação ATL.

Código 4.1 em ATL. Especificação do mapeamento lambda envolvendo redes de Petri como um helper

```
1: helper def: lambda_r2(t : PetriNet!Transition) : PetriNet!Transition =
2:   PetriNet!Transition.allInstances()->
3:    select(ts | ts.name.text = t.name + ";" + t.name + "_copy‘‘)->first();
```

Código 4.2 em ATL. Especificação do mapeamento lambda envolvendo também redes IOPT como um helper

```
1: helper def: lambda_r2(t : PetriNet!Transition) : IOPT!SynchronySet =
2:   IOPT!SynchronySet.allInstances()->
3:    select(ss | ss.master.name.text = t.name.text+'master')->first();
```

Tomando a aplicação da regra #2 para modelos M_{in} e M_{out} , temos na Figura 23, a ilustração do mapeamento λ_r2. Os conceitos que envolvem a equivalência λ_r2 continuam os mesmos, apesar de trabalharmos em um cenário de aplicação da transformação a modelos mais complexos.

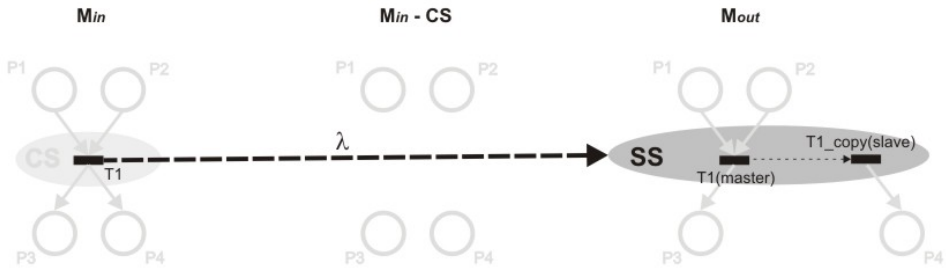

Figura 23: Mapeamento λ_r2 sobre os conceitos em uma aplicação da regra #2

Por fim, contextualizamos nas Figuras 24 e 25 o papel do mapeamento λ_r2 na relação entre conceitos sintáticos para as demais regras da transformação Splitting.

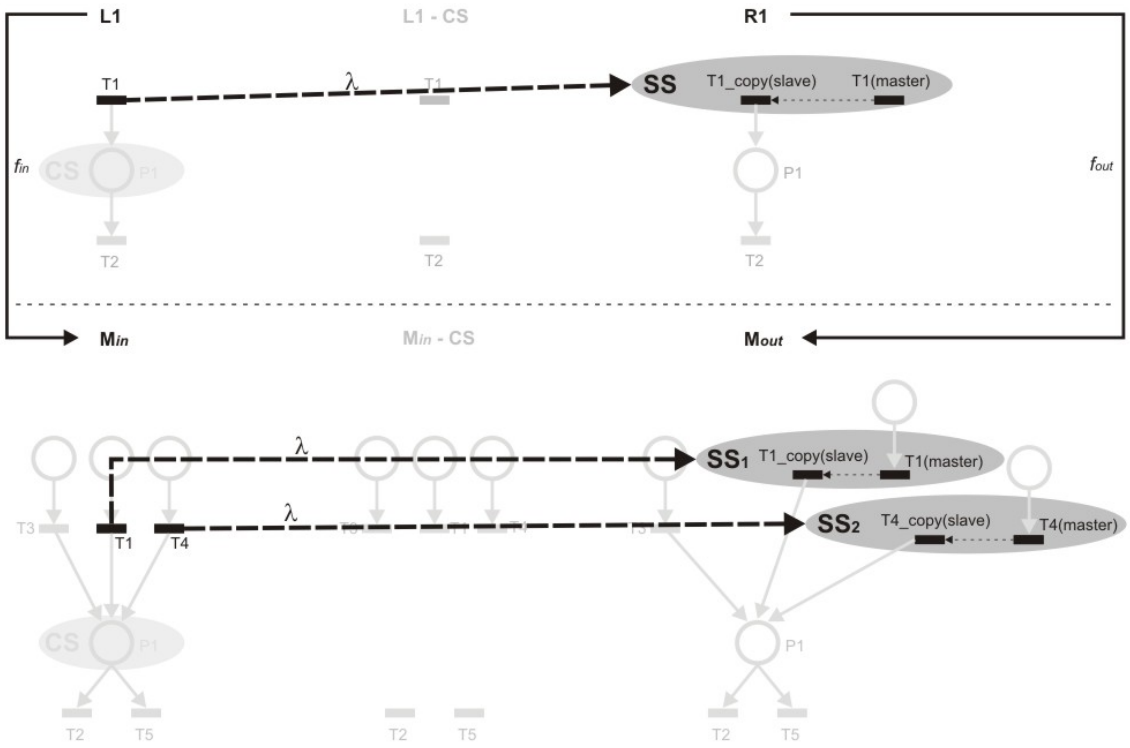

Figura 24: Contextualização do mapeamento λ_r1 para regra #1

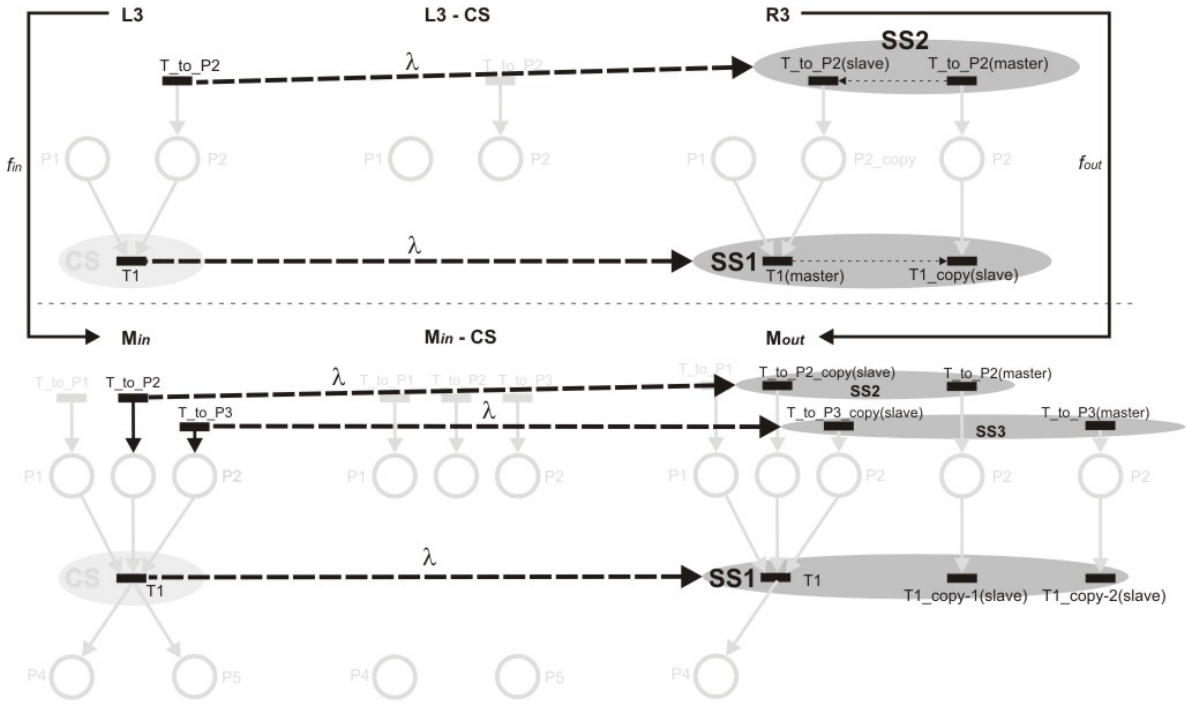

Figura 25: Contextualização do mapeamento λ_r3 para regra #3

4.2 Instanciação da MDA-Veritas Através de Modelos Redes de Petri

A instanciação seguiu o processo especificado na Figura 20 para o ator *Provedor da Solução*, permitindo reuso para verificação de qualquer transformação envolvendo M_{in} e M_{out} em redes de Petri e redes IOPT, que exercem o papel de MM_{in} e/ou MM_{out}, em que o *Cliente MDA* estivesse interessado.

Para este domínio, dizemos que uma metaclasse $mc_{in} \in MM_{in}$ ou que uma metaclasse $mc_{out} \in MM_{out}$. Exemplos de metaclasses são Place (Lugar), Transition (Transição) e outras mais definidas em Ecore. Exemplos de elementos $e \in M_{in}$ ou $e \in M_{out}$ são instâncias de lugares (P1, P2, ... , Pn), de transições (T1, T2, ... , Tm) e outras mais criadas no formato Ecore.

A transformação Tr está definida na linguagem ATL [Bezivin et al., 2003]. Temos que $LeftElements$ e $RightElements$ são regras declarativas, também chamadas de padrão fonte e destino respectivamente. Estas regras podem ser complementadas com um bloco imperativo, que executa após a aplicação do padrão destino.

Retomamos à categoria Petri que define modelos semânticos para M_{in} e M_{out}. Para encontrarmos o casamento com a abordagem MDA, reusamos resultados presentes em [Ehrig et al., 2006b] para transformações em redes de Petri, onde regras de transformação assumem a forma r_i = ($LeftElements$,$RightElements$) como um par de redes de Petri. Aplicar r_i = ($LeftElements$,$RightElements$) significa encontrar o casamento de elementos que assumem a característica de uma rede de Petri como $LeftElements$ na rede M_{in} e substituir a $RightElements$ por elementos que assumem a característica de uma rede de Petri como $RightElements$ na rede M_{out}. Para realizarmos essa substituição, precisamos conectar $RightElements$ com uma rede denominada C (contexto) produzindo M_{out}. Para formalizar esse processo, e em consonância com a abordagem algébrica iniciada no trabalho, necessitaríamos de um diagrama que representasse a definição das regras, suas substituições e as aplicações como construções algébricas.

Os esforços mais significativos foram atribuídos ao ator que faz o papel de *Provedor da Solução* e, como será observado, estão concentrados no módulo $\cong$. Contudo, o complemento dos outros módulos prescritos pela MDA-Veritas ainda é requisitado, tais como os MS_{in} mais MS_{out}, e MMS_{in} mais MMS_{out}. A seguir, apresentamos a instanciação completa de cada módulo, cobrindo cada passo do processo definido para a concretização dessa tarefa.

4.2.1 Instanciação da Atividade i: Prover regras de boa formação

M_{in} e M_{out} são checados através de regras de que utilizam um ambiente estático com descrições que os modelos precisam satisfazer a partir de sua sintaxe. Utilizamos OCL como uma linguagem de expressão para descrever um conjunto de regras e restrições semânticas para que sejam descritos por MM_{in} e MM_{out}. Decidimos optar por OCL devido aos seguintes fatos: (i) ela é parte da infra-estrutura MDA como um padrão da OMG; (ii) ela pré-define um número de tipos úteis para manipulação dos conceitos envolvidos; e (iii) existe uma especificação completa e boa quantidade de ferramentas que provêem suporte para OCL [OMG, 2011c] [KentModellingFramework, 2011] [Gentleware, 2011] [KlasseObjecten, 2011].

Adicionamos restrições importantes dadas pela definição formal de redes de Petri, apresentada no Capítulo 2.3, que desempenha a linguagem de modelagem de PIMs para esse contexto. Essas restrições não eram capturadas pelos metamodelos disponíveis providos para esse formalismo até então [Atlantic Zoo, 2011]. Porém, em ferramentas consolidadas como [Jensen, 1991] ou [Best et al., 1997] já faziam uso de algumas dessas restrições, mesmo sem a facilidade existente nos dias atuais para especificação e construção de metamodelos.

O Código 4.3 apresenta uma restrição muito simples, introduzida no metamodelo de redes de Petri como regra de boa formação a partir da sintaxe de M_{in} e M_{out}. Esta regra ainda não havia sido capturada por seu metamodelo proposto previamente. A linha 1 determina a invariante, que requer que lugares e transições sejam disjuntos em uma rede de Petri. Então, a linha 2 garante que a interseção do conjunto de lugares com o conjunto de transições deve ser vazia. Esta invariante é uma regra semântica que pode ser checada em tempo de compilação.

Código 4.3 em OCL. Exemplo de uma restrição de referência a um ambiente estático para redes de Petri

```
1: context PetriNet inv PlacesTransitionsDisjoint:
2: Place.allInstances().intersection(Transition.all Instances())->isEmpty()
```

4.2.2 Instanciação da Atividade ii: Metamodelos Semânticos para o Domínio de Verificação

Propomos o uso de metamodelos MMS_{in} e MMS_{out} para descrever a sintaxe abstrata para a definição de álgebras semânticas que possam servir como linguagem para definição de significado dos modelos, ou seja, MS_{in} e MS_{out}. A definição dessa álgebra semântica precisa ser guiada por uma teoria que seja considerada sólida para o domínio analisado, caracterizando-se como uma

atividade de instanciação manual. Em particular, para redes de Petri, a formalização desenvolvida em [Meseguer and Montanari, 1988] e já apresentada no Capítulo 2.4.4 foi escolhida para ser empregada no uso de MMS_{in} e MMS_{out}. Neste caso, MS_{in} e MS_{out} definidos por MMS_{in} e MMS_{out} para redes de Petri devem ser vistos como uma representação concreta de um grafo na categoria Petri, no qual é composto por um monóide, que satisfaz suas propriedades básicas, como identidade, fechamento e associatividade mais a propriedade comutatividade como um monóide comutativo.

Reusamos o metamodelo da solução Maude com o objetivo de incorporar conceitos particulares da representação em categorias de redes de Petri. Desta forma, decidimos instanciar todos os domínios semânticos desse trabalho em lógica de reescrita, formalismo cujo Maude implementa, generalizando MMS_{in} e MMS_{out} ao metamodelo dessa linguagem. Isto origina uma semântica translacional para essa linguagem, por essa ferramenta ser capaz de representar todas as visões semânticas que estamos interessados, ou seja, operacional, denotacional e algébrica. Além do mais, essa delegação soa como um atrativo à simplicidade da nossa técnica, já que poderemos reusar metamodelos de ferramentas de métodos formais que satisfaçam as teorias semânticas empregadas. Como ilustração, a Figura 26 apresenta um pequeno trecho desse metamodelo com construções sintáticas do Maude, tais como `Module`, `Sort`, `Operation`, `Term` e outras mais, comuns à essa linguagem.

4.2.3 Instanciação da Atividade iii: Descrever as equações semânticas

Contando com Tr, M_{in} e M_{out}, é requisitada uma investigação de uma maneira para que se possa extrair automaticamente o significado dos modelos para um domínio que permita aplicar técnicas formais para verificação de equivalência entre os modelos.

Assumindo que MMS_{in} e MMS_{out} garantem que conceitos de redes de Petri apresentados na Seção 2.3 podem ser formalizados na teoria das categorias de uma maneira genérica, MS_{in} e MS_{out} serão as instâncias desses conceitos construídos através de EqS_{in} e EqS_{out} respectivamente. Como um exemplo, podemos ver dois casos de modelos redes de Petri e IOPT descritos na Figura 27, onde o modelo na parte (a) da figura é M_{in}, instância do metamodelo redes de Petri como MM_{in} e o da parte (b) é M_{out}, instância do metamodelo IOPT como MM_{out}, representando uma rede de Petri particionada como um PIM ou um modelo para alguma plataforma específica de sistemas embarcados como um PSM. Sob o ponto de vista de MS_{in} e MS_{out}, esses modelos

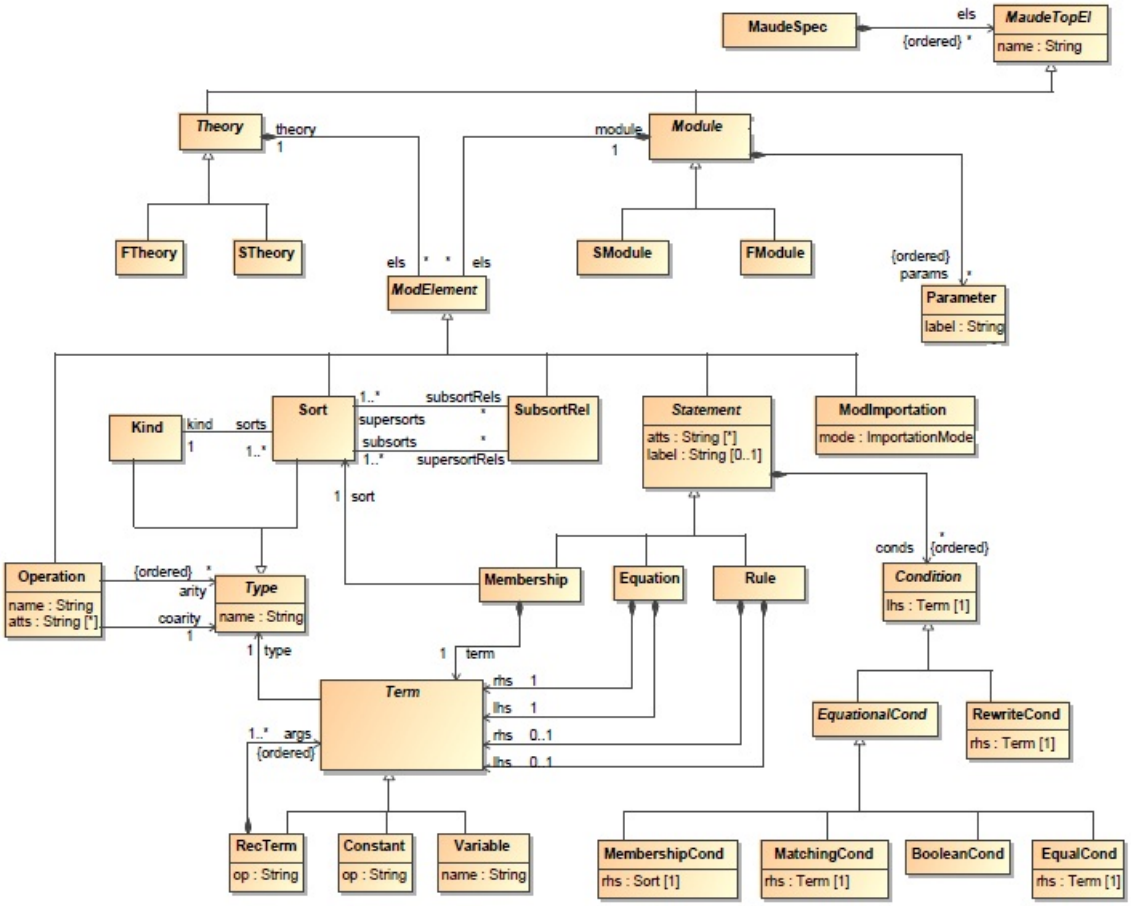

Figura 26: Fragmento do metamodelo do Maude

podem ser representados como um grafo ordinário, cujo conjunto de nós é uma estrutura algébrica gerada pelo conjunto de lugares e os morfismos são produzidos pelas transições. Dado que temos uma marca em cada lugar (`P1` e `P2`), EqS_{in} e EqS_{out} devem especificar que MS_{in} e MS_{out} sejam:

- $M_{in}\ T1 : P1 \oplus P2 \rightarrow 2'P2$
- $M_{out}\ (Comp_1\ T1 :\ P1)\ \mathbf{;}\ (Comp_2\ T1\_copy :\ P2) \rightarrow\ (Comp_2\ 2'P2)$.

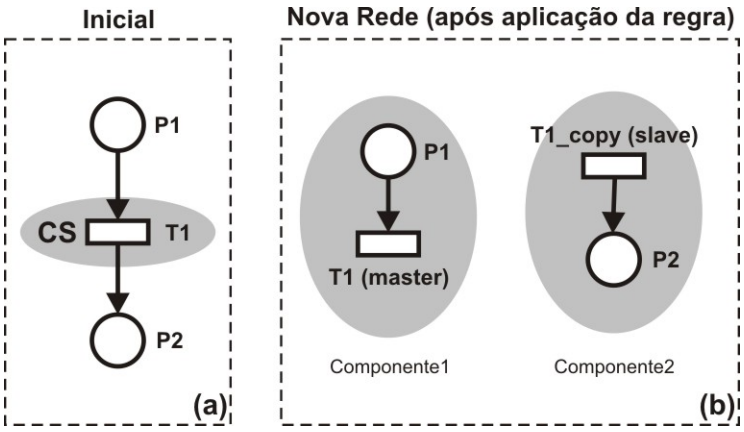

Figura 27: Exemplo de dois modelos descritos em redes de Petri

Neste caso, EqS_{in} e EqS_{out} adaptam duas operações algébricas herdadas de [Meseguer and Montanari, 1988]: $\oplus$ e **;**. O primeiro corresponde à união dos elementos e o último à concatenação de arcos como composição seqüencial. Isto significa que, na saída, as principais diferenças sintáticas observadas serão: existir dois componentes de rede fragmentados, onde não é permitida a operação de união de *multiconjuntos* sobre a marcação dos lugares `P1` e `P2` e existir uma composição seqüencial entre a transição `T1` e a nova transição inserida `T1_copy`. A *teoria das categorias* é útil para explicar esses aspectos, usando uma notação algébrica formal.

Para guiar a construção de EqS_{in} e EqS_{out}, temos em [Meseguer and Braga, 2004] idéias fundamentais de mapeamentos para a lógica de reescrita como *equações semânticas*, para que se faça uso de ferramentas como o Maude LTL Model-Checker, por exemplo.

Para incorporar EqS_{in} e EqS_{out} à arquitetura MDA-Veritas reusando padrões MDA, tivemos que lidar com algumas questões pragmáticas e formais. As primeiras dizem respeito a requisitos relacionados às linguagens e ferramentas que irão representar essas *equações semânticas*, enquanto que as últimas lidam com aspectos teóricos para garantir composição e recursão.

Focando nos aspectos pragmáticos, há três requisitos que devem ser observados na linguagem usada para representar EqS_{in} e EqS_{out}. O primeiro é que a linguagem deve permitir representar as

características implementadas no formalismo empregado pela semântica denotacional, como em nosso domínio, representando a categoria Petri. Além do mais, essa linguagem precisa representar características de linguagens declarativas corretamente, para que conceitos sintáticos sejam diretamente mapeados em seus domínios semânticos. O segundo requisito indica que a linguagem deva suportar a representação de características reais da maioria dos modelos de programação. Nos concentramos nas características dos modelos simples em redes de Petri tratados anteriormente. Isto significa que a linguagem deva permitir representar aspectos de linguagens de programação e de modelagem como seqüencialidade, concorrência, não-determinismo e uma infra-estrutura para representar estados. Por último, o terceiro requisito se relaciona com a manipulação e execução dessas *equações semânticas*, permitindo sua integração com outras ferramentas de métodos formais.

Por outro lado, há dois aspectos da computação teórica que precisam ser revisitados. O primeiro é que a linguagem escolhida para representar EqS_{in} e EqS_{out} seja completamente (ou parcialmente) composicional. O entendimento de expressões compostas como regras de transformação torna mais fácil a prova de propriedades semânticas, permitindo a tradução de peças sintáticas para metalinguagens semânticas representadas como MMS_{in} e MMS_{out}. Isto representa um desafio e uma melhora significativa porque geralmente há proposições, axiomas e teoremas que já foram desenvolvidos para essa metalinguagem. Encontrar essa linguagem composicional permite que raciocinemos usando indução sobre a gramática da linguagem de modelagem, desde que essa gramática seja descrita por um metamodelo. O segundo aspecto teórico é que a linguagem deva permitir recursão sobre a estrutura gramatical (novamente o metamodelo) da linguagem que essa estrutura interpreta, ou seja, ela deve permitir extrair o estado corrente do modelo em termos de uma representação significativa, ou seja, o *modelo semântico* através de MMS_{in} e MMS_{out}. Isto é feito carregando-se a estrutura que representa o ambiente de um modelo. Quando M_{in} é repassado ao engenho que aplica EqS_{in} e EqS_{out}, a transformação explora esse modelo a um nível de cima para baixo, criando contextos necessários para uma estrutura de ambiente, como ambientes de thread, memória, localizações e valores. Após isso, essa estrutura é repassada em uma chamada recursiva a cada um dos subtermos, e provê os termos prontos para serem analisados baseado no que se retorna do mapeamento que representa EqS_{in} e EqS_{out}.

Como vimos, EqS_{in} e EqS_{out}, segundo a ótica desse trabalho, são essencialmente transformações MDA. Neste trabalho, usaremos uma linguagem de transformação MDA e padrões para

satisfazer às questões pragmáticas e teóricas apresentadas. A linguagem ATL [Bezivin et al., 2003] emerge como a principal candidata para esta tarefa devido às características compatíveis com os paradigmas declarativo e imperativo e seu conjunto poderoso de ferramentas. Contudo, como base da recursão, apesar das técnicas formais escolhidas para representação dos domínios semânticos e dos cuidadosos mapeamentos que serão realizados a partir da sintaxe pelas *equações semânticas*, no momento não temos uma garantia que este caso de transformação seja estritamente preservadora de semântica.

4.2.4 Instanciação da Atividade iv: Especificar a representação dos modelos semânticos

Para se especificar MS_{in} e MS_{out}, adicionamos aos conceitos transformados por EqS_{in} e EqS_{out}, também uma representação do comportamento, ou *semântica dinâmica* do modelo. Em uma rede de Petri isto envolve o conceito de marcação, na qual é uma função que associa a cada lugar um inteiro não-negativo, chamado de *marca*. Para análises operacionais de redes de Petri, um estado é composto da marcação corrente, que é armazenada em uma estrutura de dados chamada de *Multiconjunto*. Suas computações são representadas pelo disparo de alguma transição habilitada dada pela marcação corrente. A operações algébricas $\oplus$ e **;** permitem a composição dessas marcações.

A *semântica dinâmica* requer uma descrição das operações em termos de alguma máquina, seja real ou abstrata. Adotamos o *framework de lógica de reescrita* como a ferramenta de unificação da semântica operacional e denotacional [Meseguer and Rosu, 2004]. Assim, na perspectiva operacional, ou de aspectos dinâmicos, MS_{in} e MS_{out} são executáveis via reescrita sem esforços adicionais. Também consideramos o suporte ferramental da lógica de reescrita e os protótipos disponíveis adequados para o escopo dessa investigação, baseando-se na existência de casos de sucesso nessa perspectiva [Chalub and Braga, 2005]. Mais especificamente, empregamos a solução Maude, através do método de reflexão [Clavel and Meseguer, 1997], para redefinir a semântica do comportamento das redes de Petri. Maiores detalhes sobre os componentes dessa implementação estão a seguir:

- Um tipo `[Marking]` (marcação) junto com os símbolos de operadores que deve possuir os seguintes operadores: `empty:` $\rightarrow$ `[Marking]`, que representa uma marcação vazia como sendo um operador caso base para a composição de marcações e `_`$\oplus$`_: [Marking][Marking]` $\rightarrow$ `[Marking]` como sendo a operação de composição de duas marcações propriamente.

- Símbolos de operadores como: `p:` $\rightarrow$ `[Marking].`
- Os axiomas de composição paralela: `∀u,v,w: [Marking]. u⊕(v⊕w) = (u⊕v)⊕w` garantindo associatividade, `∀u,v: [Marking]. u⊕v = v⊕u` garantindo comutatividade e `∀ u: [Marking].empty⊕u = u` garantindo composição com elemente neutro. Todas essas propriedades são derivadas de conceitos já presentes MMS_{in} e MMS_{out}, devendo serem instanciadas para esse domínio.

Classicamente, podemos trabalhar com dois níveis de especificação em abordagens de verificação: (i) nível de especificação de sistema, que requer uma formalização completa do sistema concorrente a ser analisado e (ii) nível de especificação de propriedades, no qual as propriedades a serem verificadas são especificadas. A técnica empregada, ao reusar o sistema Maude, permite trabalhar com esses dois níveis de especificação para os modelos ao validar transformações MDA. Reusamos a definição da lógica de reescrita, com definindo uma teoria equacional descrevendo a estrutura de estado de uma rede de Petri, correspondente ao nível de especificação de sistema e um conjunto de regras de reescrita representando transições concorrentes, no qual suas seqüências de execução combinadas originam diversos tipos de características, pertencentes ao nível de propriedades. Isto também permitiu o uso de técnicas de uma completude maior, como provadores de teoremas. A mesma teoria de reescrita definida pôde ser empregada como entrada para um verificador de modelos LTL [Eker et al., 2003]. Isto foi provido de maneira *On-the-Fly* pela ferramenta Maude. Para os casos de a propriedade especificada como uma fórmula LTL ter sido válida, a ferramenta forneceu um resultado *verdade*. Para os casos contrários, um resultado *falso* foi apresentado junto com um contra-exemplo de como essa negação pôde ser atingida. Este verificador de modelos possui uma implementação bastante eficiente, que além de reusar o alto grau de expressividade para representação de semânticas provido pelo Maude, também é capaz de lidar com o problema de explosão de espaços de estados de uma maneira similar a algumas soluções de ponta nessa área, tal como o Model-Checker Spin [Holzmann, 1997].

Representamos no Código 4.4 o módulo Maude `NET`, que descreve o comportamento da rede de Petri mostrada na Figura 27(a). Lugares são representados como operadores do tipo `Place` (linha 4). Transições são representadas como regras de reescrita (linha 7). A execução de uma regra denota o disparo de uma transição. Já no Código 4.5, temos o módulo `NET-SPLITTED`. Este módulo incorpora uma configuração, que é uma característica do Full Maude para descrever objetos concorrentes. Os tipos, lugares e regras algébricas permanecem os mesmos. Porém na

linha 8 temos a idéia de um componente de uma rede. Este componente representa um módulo particionado da rede. A comunicação entre esses componentes se dá na pelo objeto Msg definido na linha 10, que representa uma mensagem de comunicação de um canal. Após declaração de variáveis, as linhas 14-16 representam a execução da transição particionada T1, que faz a associação `T1;T1_copy(Comp1,Comp2)` mudando o estado dos dois componentes.

Código 4.4 em Maude. Representação de uma rede de Petri

```
1: mod NET is
2:  sorts Place Marking .
3:  subsort Place < Marking .
4:  ops P1 P2 : -> Place .
5:  op empty : -> Marking .
6:  op __ : Marking Marking -> Marking [assoc comm id: empty] .
7:  rl [T1] : P1 => P2 .
8: endm
```

Código 4.5 em Maude. Representação de uma rede de Petri particionada

```
1: mod NET-SPLITTED is
2:  inc CONFIGURATION .
3:  sorts Place Marking .
4:  subsort Place < Marking .
5:  ops P1 P2 : -> Place .
6:  op empty : -> Marking .
7:  op __ : Marking Marking -> Marking [assoc comm id: empty] .
8:  op Component : -> Cid [ctor] .
9:  op m:_ : Marking -> Attribute [ctor gather (&)] .
10: ops T1;T1_copy : Oid Oid -> Msg [ctor] .
11: vars Comp1 Comp2 : Oid .
12: var C : Configuration .
13: var Any : Marking .
14: rl[T1;T1_copy] : T1;T1_copy(Comp1,Comp2)
15:  <Comp1:Component | m:P1 Any> <Comp2:Component | m:Any>
16:  => < Comp1:Component | m:Any> <Comp2:Component | m:Any P2> .
17:endm
```

4.2.5 Instanciação da Atividade v: Expressar a equivalência dos modelos em redes de Petri

Como já mencionado neste trabalho, faz-se necessário adaptar o formalismo do *Verificador Formal* de acordo com o paradigma dos modelos envolvidos na transformação. A abordagem é apoiada por uma técnica formal que preenche o módulo *Verificador Formal* da MDA-Veritas: model-checking

e comparação de resultados. A aplicação desta técnica é dividida em algumas partes de complexidade considerável, inclusive fazendo uso da combinação de outras técnicas formais.

A primeira parte ainda está relacionada à execução da transformação feita pelo *Cliente MDA*, na qual, para cada regra aplicada $r_i \in Tr$, análises provêem TES entre M_{in} e M_{out}. Após isso, a técnica de verificação de equivalência é combinada com outra técnica de abordagem operacional, chamada de verificação de modelos, para desempenharem um papel central no *Verificador Formal*. Essas partes são detalhadas a seguir.

Relação de Equivalência Sintática

Estabelecemos um método para extração de uma Tabela de Equivalência Sintática TES nas transformações analisadas pela MDA-Veritas. Recorremos à uma lista de equivalências simples Π para esse domínio, conforme foi apresentada na Seção 2.3.3. Essa lista Π tem o objetivo de aproximar a definição de Tr existente a uma decomposição das regras em passos atômicos, que, formalize os questionamentos existentes sobre preservação de semântica das regras, oriente a definição da noção de equivalência e guie a geração automática de fórmulas para verificar essa equivalência em termos de comportamento dos modelos.

A Tabela de Equivalência Sintática TES é definida como um conjunto onde cada um de seus elementos será $line_i$ = (M_{in}Cell, M_{out}Cell). Esta relação está descrita no metamodelo da Tabela de Equivalência Sintática MM_{TES}, que está descrito na Figura 28. Cada elemento $line_i$ será instância da metaclasse *Line*, com M_{in}Cell e M_{out}Cell sendo resumida ao atributo conteúdo (`content`). Este atributo é uma especificação da semântica de execução dos componentes envolvidos e será detalhada mais adiante.

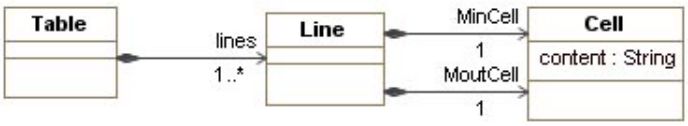

Figura 28: O metamodelo da tabela de equivalência sintática MM_{TES}

Desta forma, tomando Tr e de acordo com o processo estabelecido na Figura 29, é papel do *Provedor da Solução*: (i) identificar um conjunto Π de equivalências básicas para o domínio conforme apresentado na Seção 2.3.3; (ii) para cada regra r_i de Tr, selecione um conjunto de relações de equivalências Δ a partir de Π conforme também apresentado na Seção 2.3.3; (iii) especifique mapeamentos sintáticos Λ baseando-se nas relações Δ tal como feito na Seção 4.1.1;

(iv) defina operadores semânticos para os campos da TES através de um conjunto de funções Σ baseados nos mapeamentos oferecidos por Λ, que serão detalhados nos textos seguintes; e (v) a partir dessas especificações, fazer a geração do mapeamento `ExtractTES`. Este mapeamento produzirá a tabela TES com aplicação direta dada pelo ***Cliente MDA***, a partir dos modelos Min e $Mout$ envolvidos em Tr.

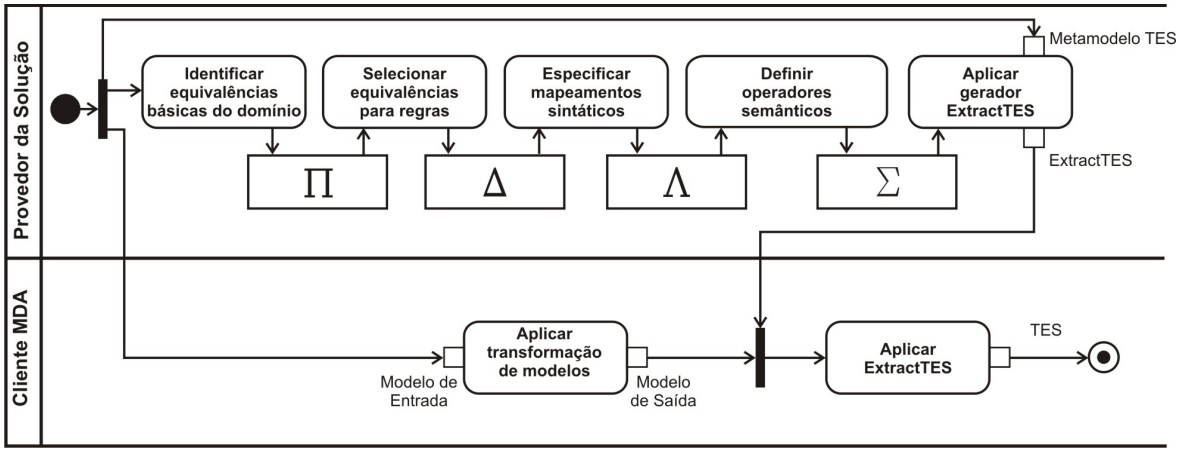

Figura 29: Processo de criação e geração da TES

O Algoritmo 1 guia todo o processo de geração do mapeamento extrator da TES, intitulado por `ExtractTES`. Este algoritmo recebe a transformação Tr, consequentemente com MM_{in} e MM_{out} associados, e o conjunto Π de primitivas básicas de equivalência sintática entre modelos. Estas primitivas possuem o intuito de guiar a geração automática ou ajudar na especificação manual de fórmulas temporais para verificação de preservação de propriedades entre modelos de interesse ao Cliente MDA. Por fim, recebe-se o MM_{TES}, que definirá as regras para a construção da Tabela de Equivalência Sintática onde sua instanciação será explorada ao longo desta seção.

Ainda no Algoritmo 1, a variável `ExtractTES` definirá o mapeamento que nos interessa na forma (MM_{in}, MM_{out}) $\rightarrow$ MM_{TES}. Além do mais, na execução do algoritmo, o seu corpo será preenchido através da operação de concatenação de String +=. Pode-se perceber o uso dessa operação quando, inicialmente, fazemos a geração do cabeçalho deste mapeamento chamando a função `Gere Cabeçalho` (linha 8), que possui sua especificação no Algoritmo 2 e que será explicado mais adiante. Para cada r_i de Tr, e a um conjunto básico de equivalências Δ_i, deve-se realizar os seguintes passos: (i) especificar um mapeamento de equivalência para a regra λ_ri (linha 10), já detalhado na Seção 4.1.1; e (ii) gerar o código do mapeamento `ExtractTES` e especificar como o conteúdo da semântica de execução dos modelos será armazenado na TES

(linhas 11-16), procedimentos estes detalhados logo a seguir.

1: **Entrada:**
2: Conjunto Π de noções de equivalência básicas para o domínio,
3: Transformação Tr e Metamodelos MM_{in} e MM_{out} associados,
4: Metamodelo Tabela de Equivalência Sintática MM_{TES}.
5: **Variáveis:**
6: Mapeamento ExtractTES: $(M_{in}, M_{out}) \rightarrow TES$.
7: **Algoritmo:**
8: ExtractTES += Gere Cabeçalho(MM_{in}, MM_{out}, MM_{TES});
9: **Para cada** r_i=($LeftElements$,$RightElements$) $\in Tr$ **faça**
10: Especifique λ_r_i: $mc_{in} \rightharpoonup mc_{out}$ usando Δ_i,
onde $mc_{in} \subseteq LeftElements$, $mc_{in} \in MM_{in}$, $mc_{out} \subseteq RightElements$ e $mc_{out} \in MM_{out}$;
11: ExtractTES += Gere Regra Models2Table_r_i(MM_{in}, MM_{out}, λ, MM_{TES});
12: ExtractTES += Gere Regra CreateLine_r_i(mc_{in}, mc_{out}, MM_{TES});
13: Especifique $\sigma M_{in}\_r_i$: $mc_{in} \rightarrow$ String;
14: ExtractTES += Gere Regra CreateMinCell_r_i(mc_{in}, σM_{in}, MM_{TES});
15: Especifique $\sigma M_{out}\_r_i$: $mc_{out} \rightarrow$ String;
16: ExtractTES += Gere Regra CreateMoutCell_r_i(mc_{out}, σM_{out}, MM_{TES}).
17: **Saída:**
18: Mapeamento ExtractTES: $(M_{in}, M_{out}) \rightarrow TES$.

Algoritmo 1: Definição da Extração da Tabela de Equivalência Sintática na MDA-Veritas

O Algoritmo 2 provê uma geração trivial do cabeçalho do mapeamento `ExtractTES` como uma transformação ATL. Essa transformação cria uma instância TES do metamodelo MM_{TES} a partir de uma instância M_{in} do metamodelo MM_{in} e uma instância M_{out} do metamodelo MM_{out}. A geração deste mapeamento se dará de acordo com as características de cada regra da transformação Tr.

Seguindo o Algoritmo 1, vemos que o próximo passo essencial para uso da solução MDA-Veritas consiste em estabelecer uma noção de equivalência sintática entre modelos para orientar a definição das proposições atômicas a serem empregadas pelo módulo *Verificador Formal*. Para cada regra r_i de Tr, deve-se definir uma função parcial $\lambda\_ri : mc_{in} \rightharpoonup mc_{out}$, com $mc_{in} \in MM_{in}$, $mc_{out} \in MM_{out}$, de rótulos que estabeleça um mapeamento. Assim, o corpo desse mapeamento λ_ri estabelece uma equivalência entre os tipos presentes fortemente baseada na lista

1: **Entrada:**
2: Metamodelos MM_{in} e MM_{out},
3: Metamodelo da Tabela de Equivalência Sintática MM_{TES}.
4: **Variáveis:**
5: String Header.
6: **Algoritmo:**
7: Header += "module ExtractTES;";
8: Header += "create TES: MM_{TES} from M_{in}: MM_{in}, M_{out}: MM_{out};".
9: **Saída:**
10: String Header.

Algoritmo 2: Gere Cabeçalho do mapeamento extractTES

Δ_i. O leitor há de observar que estas funções já foram definidas na Seção 4.1.1 durante o planejamento para essa instanciação. Os helpers ATL declarados serão agregados ao mapeamento ATL `ExtractTES`.

Exploraremos agora os Algoritmos 3, 4, 5 e 6, que são chamados pelo Algoritmo 1 no intuito de complementar o mapeamento ATL `ExtractTES`. Em cada um deles, a variável `Rule` acumulará a representação de String da regra ATL a ser gerada para a geração da regra.

Explorando o Algoritmo 3, devemos, além de `Rule`, trabalhar com as variáveis e_{in} e e_{out}, declaradas na linha 7. Seu uso essencial está na linha 15, quando e_{out} passa a ser o resultado da aplicação de λ_r_i a e_{in}. A partir daí, na linha 17, linhas da TES serão criadas associando e_{in} a e_{out}.

O Algoritmo 4 gera uma regra trivial. O essencial, é que, teremos a geração de células (linhas 10 e 11) para elementos e_{in} e e_{out}.

Finalmente, os Algoritmos 5 e 6, que possuem a mesma forma, geram regras de criação de células (linha 9). O principal detalhe que convém observar nessas regras, se dá nas linhas 10 de ambas, onde ao conteúdo da célula temos a atribuição do resultado da função σ para um dado modelo e uma dada regra. A construção desta função será discutida a seguir.

Para concluir, para armazenarmos um conteúdo nas células da TES, temos que recorrer à representação como conteúdo da formalização categórica dos elementos que compõem o domínio e contra-domínio λ_r_i. Isto é dado pela implementação dos mapeamentos σ. Esse conteúdo será utilizado na geração e especificação manual de fórmulas utilizadas por verificadores. Isto é específico para o domínio abordado e também serão construídas no capítulo a seguir para os cenários de

1: **Entrada:**

2: Metamodelos MM_{in} e MM_{out},

3: Mapeamento λ_r_i: $MM_{in}!mc_{in} \rightharpoonup MM_{out}!mc_{out}$,

4: Metamodelo da Tabela de Equivalência Sintática MM_{TES}.

5: **Variáveis:**

6: String Rule,

7: Elementos $e_{in} \in MM_{in}!mc_{in}$ e $e_{out} \in MM_{out}!mc_{out}$.

8: **Algoritmo:**

9: Rule += "rule Models2Table_r_i {";

10: Rule += " from M_{in}: MM_{in}, M_{out}: MM_{out};

11: Rule += " using {e_{out} : $MM_{out}!mc_{out}$ = OclUndefined;}";

12: Rule += " to M_{TES}: MM_{TES}!Table";

13: Rule += " do {";

14: Rule += " for(e_{in} in M_{in}.elements.oclIsKindOf($M_{in}!mc_{in}$) {";

15: Rule += " e_{out} <- $\lambda\_r_i(e_{in})$;";

16: Rule += " if (not e_{out}.oclIsUndefined())";

17: Rule += " M_{TES}.lines.append(thisModule.createLine_r_i(e_{in}, e_{out})";

18: Rule += " }";

19: Rule += " }";

20: Rule += "}".

21: **Saída:**

22: String Rule.

Algoritmo 3: Gere Regra Models2Table_r_i

1: **Entrada:**

2: Metaclasses mc_{in} e mc_{out},

3: Metamodelo da Tabela de Equivalência Sintática MM_{TES}.

4: **Variáveis:**

5: String Rule,

6: Elementos $e_{in} \in mc_{in}$ e $e_{out} \in mc_{out}$.

7: **Algoritmo:**

8: Rule += "rule CreateLine_r_i(e_{in}: mc_{in}, e_{out}: mc_{out}) {";

9: Rule += " to line: TES!Line(";

10: Rule += " minCell <- thisModule.CreateMinCell_r_i(e_{in}),";

11: Rule += " moutCell <- thisModule.CreateMoutCell_r_i(e_{out})";

12: Rule += ")";

13: Rule += "}".

14: **Saída:**

15: String Rule.

Algoritmo 4: Gere Regra CreateLine_r_i

1: **Entrada:**

2: Metaclasse mc_{in},

3: Mapeamento $\sigma M_{in}\_r_i$: $mc_{in} \rightarrow$ String,

4: Metamodelo Tabela de Equivalência Sintática MM_{TES}.

5: **Variáveis:**

6: String Rule,

7: Elemento $e_{in} \in mc_{in}$.

8: **Algoritmo:**

9: Rule += "rule CreateMinCell_r_i(e_{in}: mc_{in}) {";

10: Rule += " to cell: MM_{TES}!Cell(content <- $\sigma M_{in}\_r_i(e_{in})$)";

11: Rule += "}".

12: **Saída:**

13: String Rule.

Algoritmo 5: Gere Regra CreateMinCell_r_i

1: **Entrada:**
2: Metaclasse mc_{out},
3: Mapeamento $\sigma M_{out}\_r_i$: $mc_{out} \rightarrow$ String,
4: Metamodelo Tabela de Equivalência Sintática MM_{TES}.
5: **Variáveis:**
6: String Rule,
7: Elemento $e_{out} \in mc_{out}$.
8: **Algoritmo:**
9: Rule += "rule CreateMinCell_r_i(e_{out}: mc_{out}) {";
10: Rule += " to cell: MM_{TES}!Cell(content <- $\sigma M_{out}\_r_i(e_{out})$)";
11: Rule += "}".
12: **Saída:**
13: String Rule.

Algoritmo 6: Gere Regra CreateMoutCell_r_i

transformação abordados.

Especificação da Função de Conteúdo da Semântica de Execução

Para armazenarmos um conteúdo nas células da TES, temos que recorrer à representação como conteúdo da formalização categórica dos elementos que compõem o domínio e contra-domínio λ_r_i. Isto é dado pela implementação dos mapeamentos σ. Esse conteúdo será utilizado na geração e especificação manual de fórmulas utilizadas por verificadores. Isto é específico para o domínio abordado.

Usando as redes de Petri particionadas, vemos que transições podem ser compostas em canais através dos operadores **;** de composição sequencial e $\oplus$ de composição paralela. Por exemplo, tomando a regra #2 da Splitting como ilustração, veremos que o atributo é construído em termos dos operadores de composição da categoria Petri a elementos instância de *Transition* e *Synchrony-Set*, domínio e contradomínio de λ_r_2 respectivamente. Na semântica de execução de um canal síncrono SS, vemos que em uma $line_i$, o disparo de uma transição $t \in M_{in}$ está associada a dois sub-passos a partir de uma transição habilitada $t(master) \in SS$, com $SS \subseteq M_{out}$. Inicialmente há o disparo $t(master)$ seguido dos disparos paralelos de $\forall\, t\_copy(slave) \in SS$ [Costa, 2010]. Desta forma, para SS com n transições escravas, formalizamos sua execução em termos dos operadores da categoria Petri como: $t(master)$ **;** ($t\_copy(slave)_1 \oplus t\_copy(slave)_2 \oplus$ ... $t\_copy(slave)_n$).

Helpers ATL foram implementados para especificar os mapeamentos σ. Desta forma,

especificou-se `sigmaMin_r2`, apresentado no Código 4.6, como o nome da transição `t` (linha 2), representando o disparo desta. Já em `sigmaMout_r2`, apresentado no Código 4.7, tem-se a representação da semântica de execução do modelo em rede de Petri particionado com transição síncrona. Estas duas representações correspondem à semântica de execução presentes nos modelos semânticos dos Códigos 4.4 e 4.5.

Código 4.6 em ATL. Especificação do mapeamento sigma para modelo de entrada como um helper

```
1: helper def: sigmaMin_r2(t : PetriNet!Transition) : String =
2:    t.name.text;
```

Código 4.7 em ATL. Especificação do mapeamento sigma para modelo de saída como um helper

```
1: helper def: sigmaMin_r2(t : PetriNet!Transition) : String =
2:    t.name.text;t.name.text + '_copy';
```

Geração do Mapeamento ExtractTES e sua Aplicação para a regra #2 da Splitting

O Código 4.8 sumariza a geração do mapeamento `extractTES`, ilustrando o código da regra #2 da transformação Splitting. Na linha 2, o código especifica a criação de uma tabela `tes` a partir de dois modelos (`Min` e `Mout`). A regra principal chama-se `Models2Table_r2` (linhas 3-14), tomando uma rede de Petri para a entrada e uma rede de Petri para a saída, e para cada transição de `Min` descobre através de `lambda_r2` se existe uma transição sincronizada (`SynchronySet`) originado em `Mout`. Para cada caso existente, constrói-se uma linha na tabela através da regra `CreateLine` (linhas 14-19), onde para a primeira célula dessa linha, a regra `CreateMinCell_r2` (linhas 20-22) cria uma célula correspondente com conteúdo da ação de execução da transição, enquanto que para a segunda célula, a regra `CreateMoutCell_r2` (linhas 23-25) faz com que o conteúdo seja dado através dos operadores categóricos das transições que essa transição sincronizada agrupa.

Assim, para a definição das regras analisadas anteriormente, em especial para a ilustração da regra #2, em caso de haver um mapeamento através de `lambda_r2` de uma transição `T1` para uma sincronização `T1;T1_copy` entre uma partição mestra e uma escrava, a execução de `extractTES`, através da regra `Models2Table_r2`, produz a TES descrita na Tabela 2. Em caso de haver um número de partições escravas $n > 1$, a TES gerada teria as características apresentadas na Tabela 3.

Uso da Tabela de Equivalência Sintática na MDA-Veritas

A técnica de verificação empregada à Tr baseia-se em considerar uma transformação como cor-

reta sempre que as propriedades aplicadas aos modelos M_{in} e M_{out} apresentarem comportamentos similares. Então, corretude da transformação é sempre relativa à especificação de comportamento dos modelos transformados e, em nosso caso, não é uma propriedade absoluta da transformação. As principais justificativas para a adoção dessa técnica são: (i) é uma técnica geral, que permite cobrir vários domínios; (ii) a verificação pode ser parcial, focando apenas em um subconjunto interessante de propriedades dos modelos; (iii) provê informações sobre as características dos erros encontrados, facilitando a comparação; (iv) não requer interação após a especificação; (v) possui uma base formal sólida, integrada com representações de linguagens formais.

Código 4.8 em ATL. Sumário do mapeamento extractTES

```
1:  module extractTES;
2:  create tes : TES from Min : PetriNet, Mout : PetriNet;
...
3:  rule Models2Table_r2 {
4:   from Min: PetriNet!PetriNet, Mout: PetriNet!PetriNet
5:   using { ts : PetriNet!Transition = OclUndefined; }
6:   to table: TES!Table
7:   do {
8:    for (t in Min.pages.first().objects.oclIsKindOf(PetriNet!Transition)) {
9:     ts <- thisModule.lambda_r2(t);
10:    if (not ts.oclIsUndefined())
11:     table.lines.append(thisModule.CreateLine_r2(t,ts));
12:   }
13:  }
14: }
...
15: rule CreateLine_r2(t: PetriNet!Transition, ts: PetriNet!Transition) {
16:  to line: TES!Line(
17:       minCell <- thisModule.CreateMinCell_r2(t), moutCell <- thisModule.CreateMoutCell_r2(ts)
18:     )
19: }
...
20: rule CreateMinCell_r2(t: PetriNet!Transition) {
21:  to cell: TES!Cell(content <- thisModule.sigmaMin_r2(t))
22: }
...
23: rule CreateMoutCell_r2(ts: PetriNet!Transition) {
24:  to cell: TES!Cell(content <- thisModule.sigmaMout_r2(ts)
25: }
```

Como a verificação baseada em modelos é dependente das descrições de comportamentos possíveis de sistemas em uma maneira matemática e sem ambigüidades, em nossa técnica, através da TES, é possível definir um conjunto Φ com fórmulas φ extraídas automaticamente das infor-

M_{in}	M_{out}
T1	T1;T1_copy

Tabela 2: Exemplo de TES com número de escravos n = 1

M_{in}	M_{out}
T1	T1;T1_copy$_1$ + T1_copy$_2$ + ... + T1_copy$_n$)

Tabela 3: Exemplo de TES com número de escravos n > 1

mações presentes na TES. Além do mais, dependendo do nível de conhecimento que o *Provedor da Solução* tenha com LTL, é possível que este defina manualmente um conjunto Ψ com propriedades especificadas ψ de acordo com o entendimento do domínio, que complementem o entendimento de preservação de semântica de Tr por parte do *Cliente MDA*. Essas propriedades são construídas com operadores que referem ao comportamento dos modelos ao longo do tempo. Isto permite a definição de uma garantia de equivalência funcional entre os modelos. Finalmente, definimos o conjunto $P = \Phi \cup \Psi$ que representa todas as propriedades a serem verificadas para Tr no contexto de M_{in} e M_{out}.

P possui propriedades típicas de natureza qualitativa. Por exemplo, uma propriedade $\psi \in \Psi$, que requisita saber se o sistema poderá alcançar uma situação de deadlock. Propriedades com características de tempo real não foram incluídas no escopo desse trabalho, embora exista todo um suporte este tipo de verificação como pode ser visto em [Clavel et al., 2011].

No contexto de aplicação deste trabalho, a especificação que a TES nos provê é útil para a geração de fórmulas que definam a semântica de execução dos modelos envolvidos. Por exemplo, definição para canais de comunicação gerados nas modelos, a partir de transições em redes de Petri, após o particionamento através da Splitting.

Em nosso método de verificação definido, inicialmente temos uma especificação φ que produzirá, através de uma geração textual, duas especificações complementares φ_{in} e φ_{out} aplicáveis a M_{in} e M_{out} respectivamente. Para o domínio do canal de comunicação descrito na TES, formalizou-se, sob a assistência do *Cliente MDA* e da especificação disponível em [Costa, 2010], o seguinte conjunto de propriedades Φ sobre o comportamento de M_{out} a partir de M_{in}, considerando-se o mapeamento que existe através de λ_r_i.

- Considere φ_1 como a seguinte especificação: se a TES relaciona M_{in} e M_{out} através do particionamento de uma transição t, deveremos garantir que o canal gerado não perderá ne-

nhuma das mensagens produzidas. Desta forma, temos φ_{in} = t, com $t \in M_{in}$ representando o disparo de uma transição t pertencente a M_{in} e φ_{out} = []($t \rightarrow$ ($tMsg$ U t_copy1 /\ ... /\ t_copy$n$). Isto significa dizer que uma mensagem ($tMsg$) originada pela transição t deve permanecer no canal até que todos os escravos sejam disparados.

- Considere φ_2 como a seguinte especificação: se a TES relaciona M_{in} e M_{out} através do particionamento de uma transição t, deveremos garantir que o canal gerado não poderá compartilhar uma dada mensagem com o submodelo que contenha uma transição master e um submodelo que contenha uma transição slave ao mesmo tempo. Desta forma, temos φ_{in} = t, com $t \in M_{in}$ representando o disparo de uma transição t pertencente a M_{in} e φ_{out} = []¬(t /\ $tMsg$ /\ t_copy). Isto significa dizer que, em caso de haver uma sincronização entre as transições t e t_copy, uma mensagem $tMsg$ não pode ser gerada devido à simultaneidade do disparo das transições.

A Tabela 4 sumariza as fórmulas geradas como a partir da TES gerada como conjuntos Φ_{in} e Φ_{out} a serem utilizadas na verificação de propriedades do canal de comunicação originado pela Splitting.

Propriedade	φ_{in}	φ_{out}
φ_1	T1	[]($T1$ -> ($T1Msg$ U $T1\_copy$)
φ_2	T1	[]¬($T1$ /\ $T1Msg$ /\ $T1\_copy$)

Tabela 4: Exemplo de TES com número de escravos n = 1

Comparação por Equivalência entre Propriedades

Para a instanciação da MDA-Veritas no domínio abordado, ao construirmos o *Verificador Formal*, partimos da definição que, para se verificar se M_{out} gerado a partir de Tr é comportamentalmente equivalente a M_{in}, M_{out} deve preservar as mesmas propriedades de M_{in} de acordo com uma relação definida pela TES mais outras propriedades complementares que sejam de interesse. Essa é a nossa definição mais abstrata de equivalência $\cong$ entre modelos nesse domínio. Esta verificação pode ser obtida através da aplicação de verificação de modelos em ambos os níveis de acordo com um conjunto de fórmulas P com uma análise de similaridade entre os resultados. Dada uma propriedade p que pode se enquadrar como $\varphi \in \Phi$ ou $\psi \in \Psi$, utilizando-se a TES, devemos ter $(M_{in} \vDash p_{in}) \Rightarrow (M_{out} \vDash p_{out})$ e $(M_{in} \nvDash p) \Rightarrow (M_{out} \nvDash p)$, com $CE_{in} = CE_{out}$ como contra-exemplos (denotados por CE) produzidos por $(M_{in} \nvDash p)$ e $(M_{out} \nvDash p)$ respectivamente. CE_{in} e

CE_{out} devem respeitar as *leis de equivalência* definidas na TES para arcos entre estados e leis de similaridade entre caminhos. Além do mais, devemos ter $CE_{in} \subseteq CE_{out}$ ou $CE_{out} \subseteq CE_{in}$ no caso de M_{out} representar uma concretização ou abstração de M_{in}. Isto resume a formalização de $\cong$ como verificação de equivalência, a ser abordada no restante do trabalho.

Como um indicador de qualidade da técnica apresentada, deve ser possível caracterizar os problemas semânticos encontrados. Isto quer dizer que devemos ter um entendimento particular das condições de semântica dinâmica e verificação formal que devem ser satisfeitas em M_{in} e M_{out} através do conjunto P. Essas propriedades apresentam-se através de condições sobre estruturas de Kripke (Labelled Transition Systems) que descrevem sistemas concorrentes e deverão ser construídas através de combinações desses operadores para representar comportamentos complexos em sistemas concorrentes. A essência da técnica decorre em função da garantia de composição entre fórmulas de lógica temporal na ferramenta de *Maude model-checking*. Assim, as respostas para a verificação de uma propriedade semântica são providas através de um valor verdade atômico e em caso de negação, com a presença de um contra-exemplo que vá contrário aos resultados esperados. As equivalências estão acordo com espaços de estados dos modelos, vistos como uma categoria de computações, restritas ao tipo `CONFIGURATION` provido pela ferramenta Maude.

Além do mais, necessita-se verificar se as propriedades especificadas permitem cobrir a equivalência de semântica formal necessária. Isto significa dizer que, as propriedades escolhidas precisam satisfazer a um subconjunto significante de comportamentos esperados para os modelos. A detecção dessa cobertura vai de acordo com o que é percorrido da estrutura do grafo de ocorrência dos modelos, que descrevem, em todas as instâncias, o comportamento de um modelo.

Finalmente, desejamos que a técnica possua bons níveis de automação. Preferimos fazer uso de técnicas de verificação ao invés de técnicas de provas em função dessa necessidade. Espera-se que o usuário da solução deva apenas especificar o modelo de entrada e aplicar a transformação. A depender de suas intenções, sua intervenção na especificação das propriedades LTL de verificação também será útil.

As classes de problemas semânticos existentes para este tipo de transformação vão de acordo com a especificação de propriedades de sistemas concorrentes. Elas devem ser satisfeitas para as métricas conforme a especificação. Estas propriedades como métricas não podem ser formalizadas para um contexto geral. Assim, consideramos como apresentação de cada métrica, uma formalização na linguagem LTL para propriedades observáveis nos sistemas, conforme é empregado no

trabalho:

1. Propriedade Alcançabilidade (atingível): <>p
   Um dado estado ou situação é atingível.

2. Propriedade Segurança (nada de errado ocorrerá): [] p
   Dentro de determinadas condições, uma dada situação nunca irá ocorrer.

3. Propriedade Vivacidade (certeza de resposta): [](p -> <>q)
   Dentro de determinadas condições, uma dada situação irá ocorrer.

4. Propriedade Equidade (justiça): []<>p
   Dentro de determinadas condições, uma dada situação irá ocorrer recorrentemente.

5. Propriedade Liberdade de Deadlock (nunca trava): []X verdade
   Para qualquer estado, existe sempre um sucessor.

Para refinarmos mais ainda e aumentarmos a precisão dessa noção de propriedades, que compõem o conjunto P através de Psi como fórmula não geradas automaticamente, reusamos padrões de especificação de propriedades para LTL, conforme apresentados no Capítulo 2.4.5. As propriedades de corretude desempenham uma importância fundamental pelo fato de serem usadas para especificação de equivalência entre modelos.

Instanciação do Verificador Formal para Verificação de Equivalência

Retomando as definições apresentadas nas atividades anteriores, tendo a especificação de comportamento de redes de Petri dada pela *teoria das categorias*, o comportamento deve ser representado através de um LTS, como um grafo cujas arestas são a história das marcações alcançadas na rede e arcos são as transições. As arestas são objetos, preservando a estrutura de *monóide comutativo* e os arcos são vistos como *morfismos*, preservando as propriedades algébricas dadas pela *teoria das categorias*. Mostraremos então como podemos verificar se os estados dos modelos são equivalentes para um dado contra-exemplo em uma fórmula. Após isso, poderemos raciocinar através de algoritmos para concluir que aquela instância particular da transformação preservou corretamente o comportamento.

Considere ainda os modelos apresentados na Figura 27. Analisando ambas as redes, das partes (a) e (b), percebemos que ambas possuem números diferentes de componentes e de transições. Por esta razão, tiramos vantagem do fato de que ambas continuam tendo o mesmo número de lugares se

considerarmos a união de todas as marcações dos componentes, ou um sub-conjunto de interesse de cada uma. Em nossa notação para as redes de Petri fragmentadas pela transformação, temos que marcações de um componente específico são representadas por $Comp_i\ m$, onde i é o índice do componente e m representa o *multiset* que armazena a marcação. Estabelecemos nossa relação de equivalência na seguinte definição:

- $\bigoplus_{i=1..n} Comp_i\ m \in M_{in} \subseteq \bigoplus_{j=1..n} Comp_j\ m \in M_{out}$ ou
  $\bigoplus_{j=1..n} Comp_j\ m \in M_{out} \subseteq \bigoplus_{i=1..n} Comp_i\ m \in M_{in}$.

No exemplo da Figura 27, ambos modelos possuem dois estados: antes (s_0 e s'_0 respectivamente) e depois (s_1 e s'_1 respectivamente) o disparo das transições. Nossa relação de equivalência é definida pela composição de todas as redes componentes do sistema. Isto é descrito da seguinte maneira:

- $s'_0 = \bigoplus_{j=1,2} Comp_j\ m = (Comp_1\ P1)\ \oplus\ (Comp_2\ P2) = P1 \oplus P2 = s_0$.
- $s'_1 = \bigoplus_{j=1,2} Comp_j\ m = (Comp_1\ \emptyset)\ \oplus\ (Comp_2\ 2'P2) = 2'P2 = s_1$.

Na primeira linha, observamos a equivalência do estado s'_0 gerado de M_{out} com o estado s_0 gerado de M_{in}. O estado s'_0 será a composição de todas as marcações dos componentes. Simplificando para o exemplo da figura, temos que o primeiro componente fica com a marcação $P1$ e o segundo com a marcação $P2$. Essa composição é exatamente o estado s_0, provando a equivalência. Análise similar ocorre também com a equivalência dos estados s'_1 de M_{out} e s_1 de M_{in}.

Após estabelecida TES, estamos prontos para verificar se propriedades importantes de M_{in} são preservadas em M_{out} após a transformação Tr. Para ilustrar o uso de verificação de modelos em nosso exemplo, verificaremos se a propriedade de *vivacidade*, ou a falta dela, é preservada no modelo analisado. *Vivacidade* garante que, em um modelo, todos os seus estados possuam marcações com pelo menos uma transição habilitada. Especificamos *habilitada* (enabled) pela garantia de que existe pelo menos uma marca `P1`:

- $s' = Comp_1\ m \subseteq P1 = P1 = s \models enabled = true$.

No exemplo da Figura 27, *vivacidade* é especificada usando LTL. Observe que essa função pertence ao subconjunto Psi de P, pois não é derivada automaticamente. Começamos a partir dos estados iniciais de ambos modelos (s_0 e s'_0), usando o operador [] (globalmente). Submetemos então para uma ferramenta abstrata de verificação de modelos:

- modelCheck(s_0, $[]enabled$) e modelCheck(s'_0, $[]enabled$).

Esta verificação de equivalência, através de verificação de modelos, produz o seguinte resultado para s_0:

- Solução: Falso

  Contra-exemplo:

  0: $P1 \oplus P2$

  1: $2'P2$

  $\langle$Não há ciclos e nem mais estados a explorar$\rangle$.

Com esta verificação, é requisitado um resultado negativo para s'_0 e uma equivalência entre os estados que compõem o contra-exemplo:

- Solução: Falso

  Contra-exemplo:

  0': $(Comp_1\ P1) \oplus (Comp_2\ P2)$

  1':$(Comp_1\ \emptyset) \oplus (Comp_2\ 2'P2)$

  $\langle$Não há ciclos e nem mais estados a explorar$\rangle$.

Desde que a mesma avaliação seja obtida e exista uma relação de equivalência entre estados provenientes de ambos contra-exemplos conforme provado anteriormente, concluímos que M_{out} preserva o mesmo comportamento com respeito à propriedade semântica de *vivacidade* de M_{in}.

4.3 Modelos Semânticos para IOPTs

Conforme apresentado no Capítulo 2.3, as IOPTs são uma extensão mais complexas das redes de Petri. Seu propósito em nosso trabalho é de representar modelos específicos de plataforma (PSMs). Apesar da instanciação realizada na Figura 21 servir também para o contexto de uso das IOPTs, devido ao maior número de plataformas e de semânticas de execução existentes, esta seção apresenta algumas considerações adicionais em relação à atividade de instanciação da MDA-Veritas. As principais são relativas às atividades iv e v.

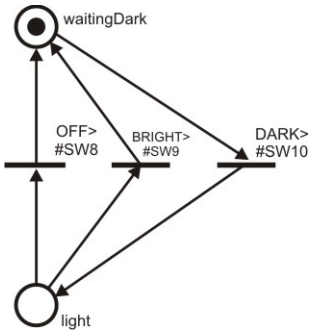

Figura 30: Fragmento de um modelo de controle residencial

4.3.1 Instanciação da Atividade iv: Especificar a representação dos modelos semânticos

As IOPTs adicionam diversas anotações aos nós e módulos das redes de Petri. Transições possuem eventos de entrada e de saída associados, além de guardas como funções dos valores de sinais de entrada. Esses modelos específicos de plataforma também devem resolver conflitos através de anotações de prioridade a cada transição. Lugares possuem uma anotação de limite especificando o número máximo de marcas de cada lugar, que é um dado importante para a geração de código. Também há ações externas condicionais nos sinais de saída, onde a condição é uma função a partir da marcação do lugar.

A Figura 30 apresenta um exemplo muito simples de uma rede IOPT para representar um controlador de luminosidade em uma residência. Há uma marca no lugar `waitingDark` enquanto o ambiente ainda está claro. Quando escurece (`DARK`), ele passa para o lugar que acende as luzes do ambiente (`light`). A partir daí, para retornar ao estado das luzes desligadas, pode ser acionado através de `OFF` ou detectar luz ambiente através de `BRIGHT`. O código a ser gerado para essas redes deve preservar as características existentes dos modelos PIMs como redes de Petri, conforme apresentadas anteriormente, mais as novas características das IOPTs, tais como eventos (OFF, BRIGHT and DARK) e sinais (SW8, SW9 and SW10).

O Código 4.9 representa os trechos mais importantes do *modelo semântico* para a rede da Figura 30. Ele é fruto de uma derivação de *modelo semântico* como representação algébrica para IOPT, reusando a estrutura básica de grafos das redes de Petri. De acordo com a Figura 21, percebe-se que pode haver diversos tipos de modelos e equações semânticas, a depender da plataforma de execução onde os modelos serão implantados. Desta forma, estes modelos descritos em Maude já apresentam características que os diferem de outros modelos semânticos que já foram apresentados ao longo deste trabalho. Para essa representação algébrica, ele descreve os

tipos existentes (linha 2), os eventos existentes (linha 3), os sinais existentes (linha 4), e os lugares existentes (linha 5). Outras construções responsáveis pela garantia de corretude da definição serão discutidas mais adiante. Finalmente, temos a representação de uma transição (linha 6), chamada de `turnLightOn` que tem como pré-condição o evento `DARK`, o sinal `SW10` e a marca que deve existir em `waitingDark`, e não produz evento de saída (`idle`), nenhum sinal de saída (`noSignal`) e uma marca para o lugar `light`.

Código 4.9 em Maude. Trecho de modelo semântico de uma rede IOPT para um controlador residencial

```
1: mod DOMOTICS-IOPT-NET-CONFIGURATION ...
...
2:    sorts Event EventSet Signal SignalSet PlaceMarking NetMarking IOPT .
3:    ops OFF BRIGHT DARK : -> Event . op idle : -> Event .
4:    ops SW8 SW9 SW10 : -> Signal . op noSignal : -> Signal .
5:    ops waitingDark light : -> PlaceMarking . op empty : -> PlaceMarking .
...
6:  rl [turnLightOn] : {DARK} + [SW10] + (waitingDark) => {idle} + [noSignal] + (light) .
7: endm
```

As redes IOPT geralmente tem a semântica do *maximal step* (*passo máximo*), conforme apresentado na Seção 2.3. Isto quer dizer que sempre que uma transição esteja habilitada e a condição externa associada seja verdadeira (o evento de entrada e o sinal de entrada sejam ambos verdadeiros), a transição é disparada. Um passo de uma rede IOPT é máximo quando nenhuma transição adicional possa ser disparada sem que se torne conflitante com alguma transição no *maximal step* escolhido. Dessa forma, definimos uma ocorrência de um passo de uma rede IOPT e a marcação sucessor respectiva. Esses modelos se desenvolverão através dos disparos de *maximal steps* sucessivos ou através dos passos de escolha como apresentados anteriormente para modelos PIM, chamados de *interleaving*.

Os seguintes fragmentos de código possuem uma representação básica para a tradução de um modelo IOPT para uma especificação em lógica de reescrita através do Maude como o *modelo semântico*. Reusamos construções de gramática regular para o caso de preenchimento de acordo com o número de elementos de um modelo específico. Esse controle foi necessário para ajudar na explicação do código a ser gerado devido ao aumento significativo de complexidade de um código IOPT como PSM quando comparado ao de uma rede de Petri como um PIM. No Código 4.10, a linha 1 possui a declaração do módulo apto a representar a rede IOPT. A linha 2 define os tipos existentes para essa especificação. Da linha 3 até a linha 5 temos a declaração dos nomes para os

tipos `Event`, `Signal` e `PlaceMarking` respectivamente e as operações de identidade correspondentes, ou seja, `idle`, `noSignal` e `empty`. Finalmente, a linha 6 é a operação básica de combinação de `PlaceMarking`s que segue os princípios da representação baseada em monóides comutativos das redes de Petri com as propriedades correspondentes.

Assim, para o Código 4.11, temos que os elementos de `Event`, `Signal` e `PlaceMarking` são compostos, das linhas 7 até a 9, em `EventSet`, `SignalSet` e `NetMarking` respectivamente. A composição dessas estruturas maiores é chamada de um tipo IOPT, conforme a linha 11.

Código 4.10 em Maude. Descrição de uma Configuração para um Modelo Semântico IOPT

```
1: mod name–IOPT–NET–CONFIGURATION ...
...
2:    sorts Event EventSet Signal SignalSet PlaceMarking NetMarking IOPT .
3:    ops (Event_names)* : –> Event . op idle : –> Event .
4:    ops (Signal_names)* : –> Signal . op noSignal : –> Signal .
5:    ops (PlaceMarking_names)* : –> PlaceMarking . op empty : –> PlaceMarking .
6:    op __ : PlaceMarking PlaceMarking –> PlaceMarking [assoc comm id: empty] .
...
```

Código 4.11 em Maude. Composição em IOPTs

```
...
7:    op {(_)*} : (Event)* –> EventSet [ctor] .
8:    op [(_)*] : (Signal)* –> SignalSet [ctor] .
9:    op (())* : (Place)* –> NetMarking .
10:   op noState : –> [IOPT] .
11:   op _+_+_ : EventSet SignalSet NetMarking –> IOPT .
```

Finalmente, no Código 4.12, as transições são representadas como regras de reescrita de uma configuração de uma rede IOPT (linha 12) para outra configuração de rede IOPT (linha 13).

Código 4.12 em Maude. Regra de reescrita como transição IOPT

```
    (
12:   rl [Rule_name] : {(Event_names)*} + [(Signal_names)*] + ((Place_names)*)
13:    => {(Event_names)*} + [(Signal_names)*] + ((Place_names)*) .
    )*
```

Representação da Semântica Maximal Step

Para darmos uma representação semântica translacional de formalismos que estendem as redes de Petri com outras semânticas de execução, necessitamos alterar a semântica de *interleaving* original do Maude. A estrutura principal dos módulos a serem produzidos por essa atividade será

descrita no Capítulo 5. Apenas como exemplo, mostramos um trecho no Código 4.13 que redefine a semântica de execução do sistema Maude para este domínio, de acordo com a especificação das redes IOPT.

Código 4.13 em Maude. Regra Maximal Step

```
1:  rl [maximal-step] : T:Term =>
2:   maxStep(T:Term, applicableRules(T:Term, rules, module)) .
 ...
3:  eq maxStep(T:Term, (rl X:Term => Y:Term [label(Q:Qid)] .) RS:RuleSet) =
4:   maxStep(getTerm(metaApply(module, T:Term, Q:Qid, none, 0)), RS:RuleSet)
5:  eq maxStep(T:Term, none) = T:Term .
```

As linhas 1 e 2 apresentam a definição básica da regra `maximal-step` para rede IOPT. Um termo `t` é reescrito através da aplicação de todas as regras permitidas a um dado módulo. Linhas 3 e 4 detalham o casamento de uma dada regra se aplicável e a chamada da operação intrínseca `metaApply` do módulo do Maude META-LEVEL. A linha 5 representa o caso em que todas as regras aplicáveis já tenham sido processadas e retornam o termo corrente.

Espaços de Estados Gerados por Diferentes Modelos Semânticos IOPT

Consideremos a Figura 31. Ela apresenta dois exemplos simples de redes IOPT. Utilizaremos estes exemplos para demonstrar os tipos de modelos semânticos construídos e extraídos para atender às necessidades do *cliente MDA* na verificação da transformação *Splitting*. Em 31(a) temos um modelo centralizado, a ser executado em apenas uma plataforma, já em 31(b) temos o modelo com dois módulos, decorrentes de uma função de particionamento. Para esses modelos, além das características dos módulos executarem segundo as filosofias *maximal step* ou *interleaving*, ilustra-se, através da funcionalidade de gerar os espaços de estados da nossa solução, as seguintes características adicionais de plataforma:

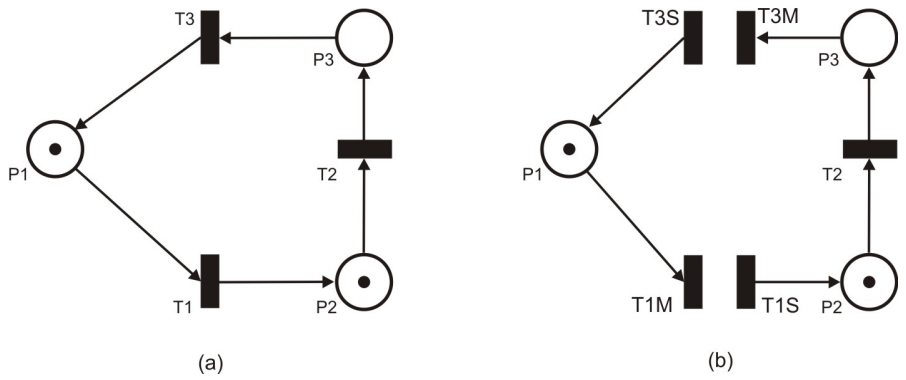

Figura 31: Dois modelos a serem instanciados com diferentes semânticas IOPT

1. Cadastros de sinais e eventos. Nesse caso, acontecem as seguintes variações:

   (a) *IOPT com Semântica Maximal Step e Eventos*. Por padrão, a rede IOPT é maximal step. Contudo, devido à presença de eventos cadastrados nas transições, e por se ter que a ocorrência de um evento pode acontecer a qualquer momento por depender do ambiente, tem-se que as características dessa rede IOPT pode tender à semântica de *interleaving*. A Figura 32 ilustra esse caso para o modelo da Figura 31(a) tendo `EvT1` como evento para `T1` e `EvT3` como evento para `T3`.

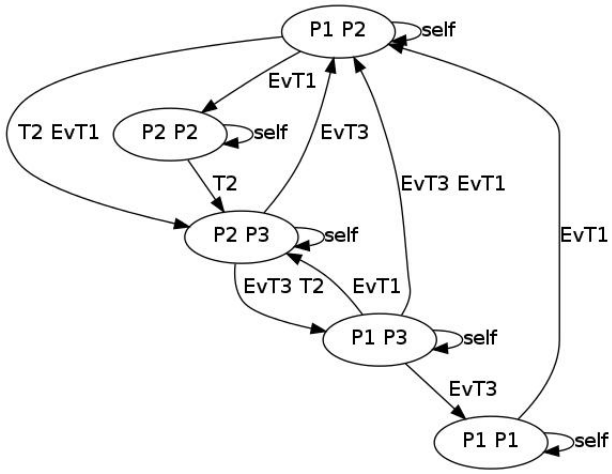

Figura 32: Espaço de estados para modelo centralizado com semântica maximal step

   (b) *IOPT com Semântica Maximal Step e Eventos Exclusivos*. Tem-se a mesma justificativa do caso anterior, porém, tem-se que a ocorrência de não apenas um evento, mas de um conjunto qualquer de eventos pode acontecer a qualquer momento por depender do ambiente. Isso está ilustrado na Figura 33 como espaço de estados para o modelo Figura 31(a). Também tende-se à semântica de *interleaving*. Esse caso representa uma discretização da leitura do ambiente, geralmente ocorrendo intervalos maiores entre os tempos de leitura. Um exemplo disso ocorre quando comparando os grafos das Figuras 32 e 33, percebe-se a ausência do arco do entre os estados (P1 P3) para (P1 P2) na Figura 33, que representaria a ocorrência simultânea dos eventos `EvT1` e `EvT3`.

2. Modelos de comunicação e clock entre módulos. Referem-se a modelos particionados com um número de módulos maior ou igual que 2. A Figura 31(b) pode ser usada como exemplo, tendo que a comunicação implica na ocorrência do evento `EvT1` e do sinal `SiT1` para a transição `T1` e do evento `EvT3` e do sinal `SiT3` para a transição `T3`:

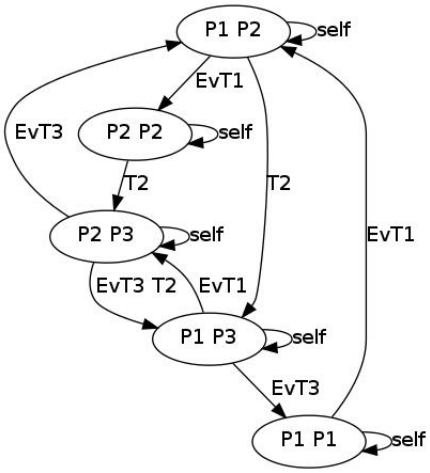

Figura 33: Espaço de estados para maximal step e exclusividade nos eventos

(a) *IOPT Síncrono com Clock Global*. Vários módulos que possuem um tick global que os sincronizam. A semântica interna é *maximal step* para cada um dos módulos e a comunicação entre eles se dá por canais síncronos. A Figura 34 representa esse caso para os dois módulos, onde o tick global é representado como arcos *maximal step* entre os estados do sistema.

3. *Mensagens Síncronas e Módulos Assíncronos*. Vários módulos particionados que a cada tick global tem-se o acionamento de apenas um desses módulos. A comunicação se dá por canais síncronos, sem estados intermediários na comunicação, e a execução local de cada um desses módulos está de acordo com a semântica de *maximal step*. A Figura 35 representa esse espaço de estados. Percebe-se que a complexidade do comportamento do modelo aumenta bastante comparando com o caso anterior devido aos módulos terem seu comportamento ditado por clocks independentes.

4. *Semântica Interleaving e Mensagens Assíncronas*. Vários módulos particionados que a cada tick global tem-se o acionamento de apenas um desses módulos. Pode ocorrer consumos de mensagens de forma assíncrona e a execução local de cada um desses módulos estará de acordo com a semântica individual de *interleaving*. O espaço de estados da Figura 36 demonstra essa existência. Esse modelo é bastante útil para testar capacidades assíncronas de comunicação entre os módulos, contudo, se distancia um pouco da realidade do nosso cenário de microcontroladores devido ao fato da semântica de *maximal step* ser a mais apta a representar a concorrência existente nesses dispositivos.

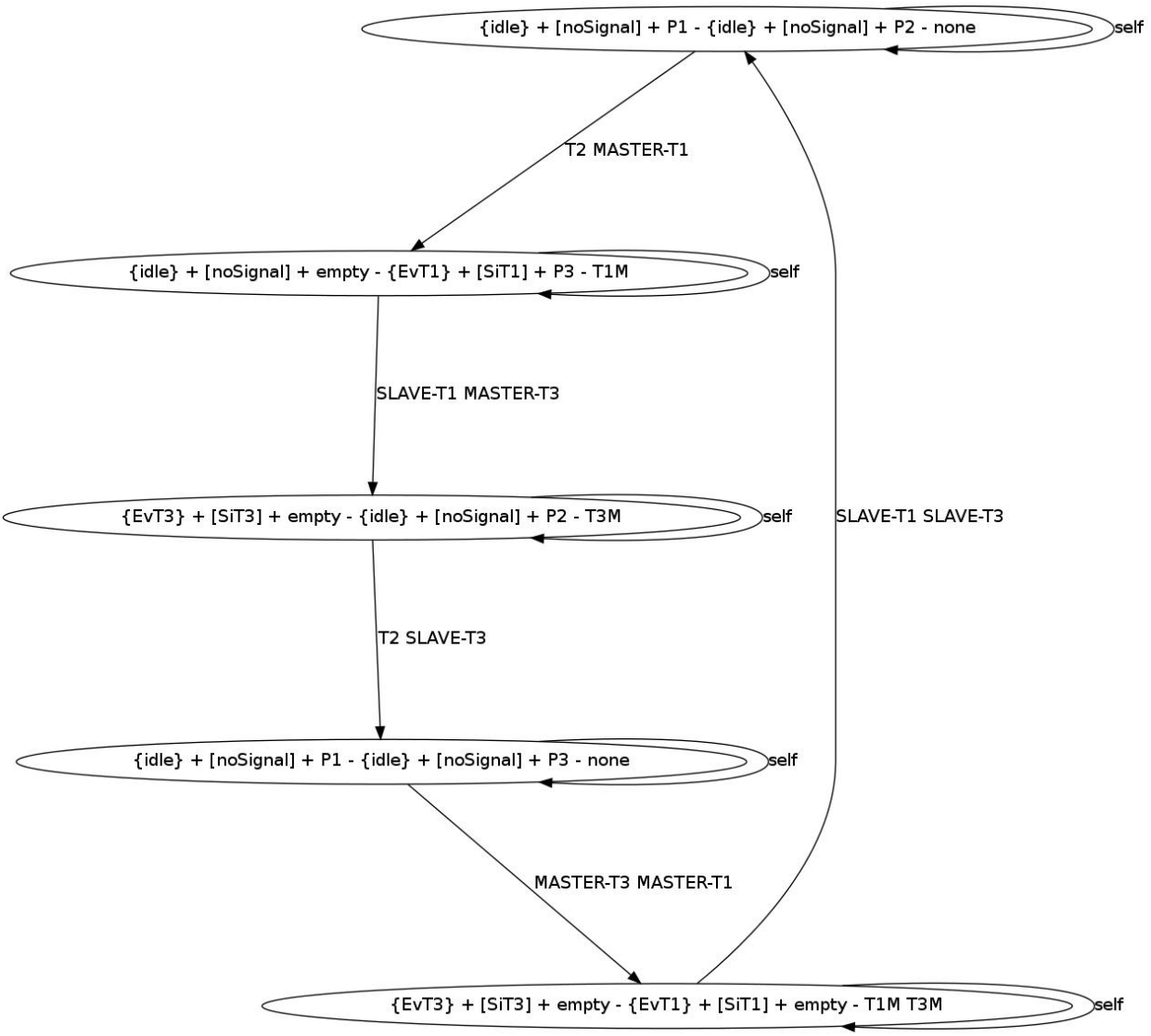

Figura 34: Espaço de estados para módulos com um único clock global

5. *Globalmente Assíncrono e Localmente Síncrono (GALS)*. Vários módulos particionados que a cada tick global tem-se o acionamento de apenas um desses módulos. Podem ocorrer consumos de mensagens de forma assíncrona e a execução local de cada um desses módulos está de acordo com a semântica de *maximal step*. Esse tipo de semântica é a de maior interesse do nosso *cliente MDA*, pois permite a integração via várias tecnologias de comunicação dos microcontroladores disponíveis atualmente. Percebemos pelo espaço de estados da Figura 37 que sua estrutura é bem mais complexa comparada aos demais. Muitas situações reais exigem essa complexidade.

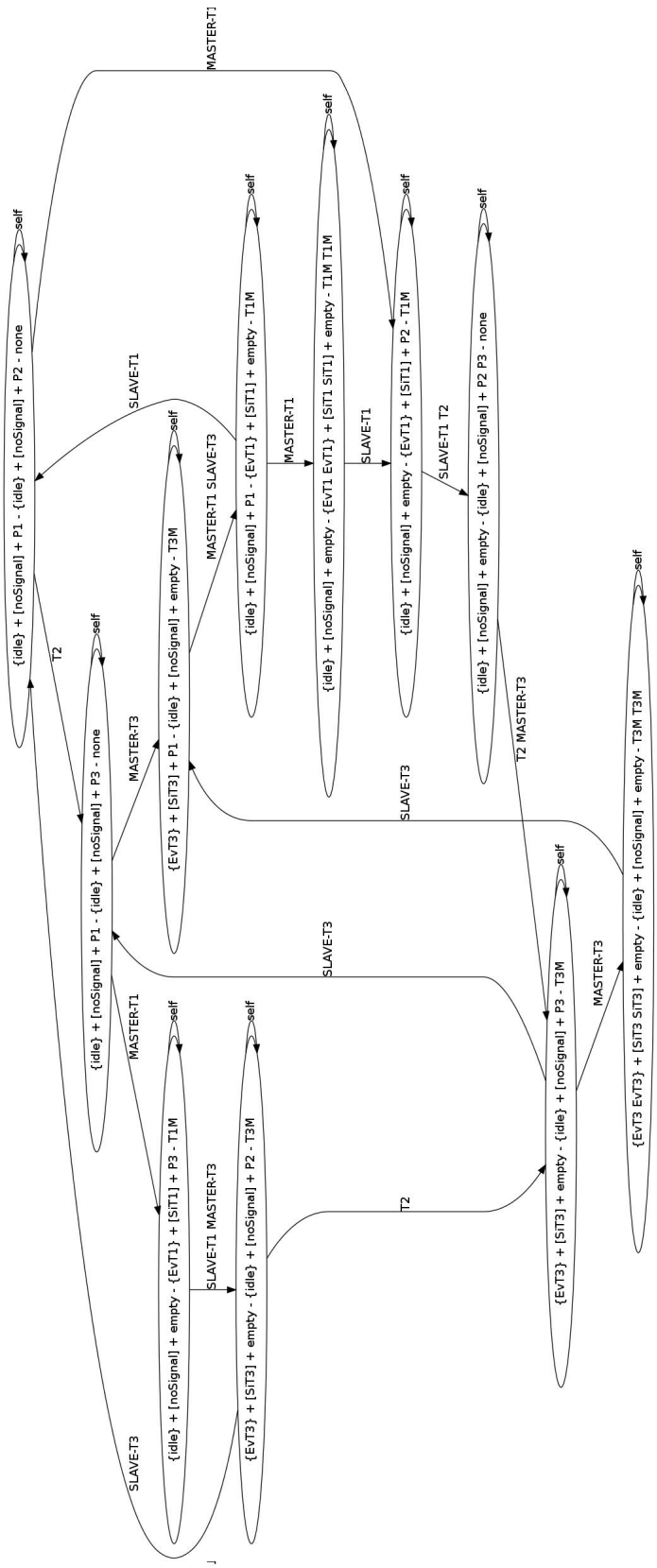

Figura 35: Espaço de estados para módulos com clocks independentes

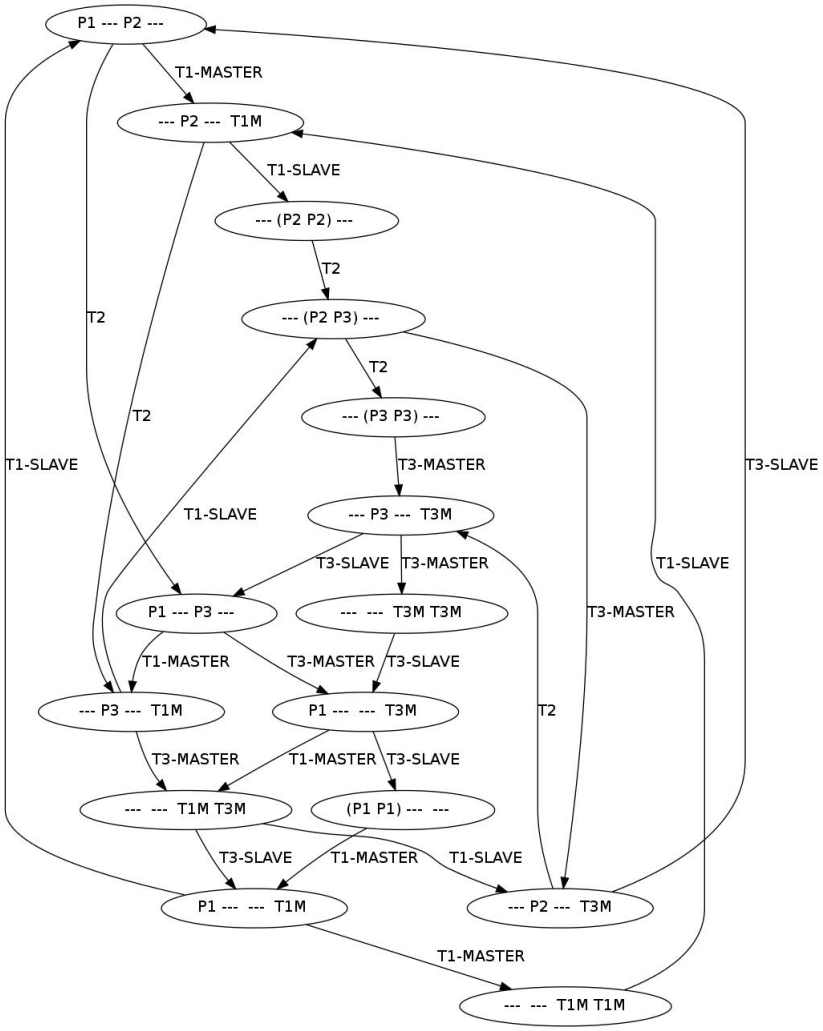

Figura 36: Espaço de estados para módulos interleaving e com troca de mensagens assíncronas

4.3.2 Instanciação da Atividade v: Expressar a equivalência dos modelos em redes IOPT

Nesta seção, revisitamos cada passo definido para expressar equivalência em redes de Petri e apontamos as modificações existentes para as redes IOPT. A maioria das questões são pontuais e envolvem ou diferenças sintáticas entre as linguagens, ou aspectos relativos às propriedades de plataformas específicas de sistemas embarcados.

Sobre as relações de equivalência sintáticas disponíveis para este domínio, a Seção 4.1.1 já apresentou as diretrizes de como redefinir as funções λ_ri incorporando conceitos de ATL. As ilustrações continuam aplicáveis para este cenário, mas consideramos a metaclasse *SynchronySet* do metamodelo IOPT, e não existente na linguagem redes redes de Petri, como fundamental em estabelecer o paradigma de comunicação das várias plataformas existentes para modelos PSM. Retomamos aqui o Código 4.14, já discutido na Seção 4.1.1, apenas com o propósito ilustrar essas

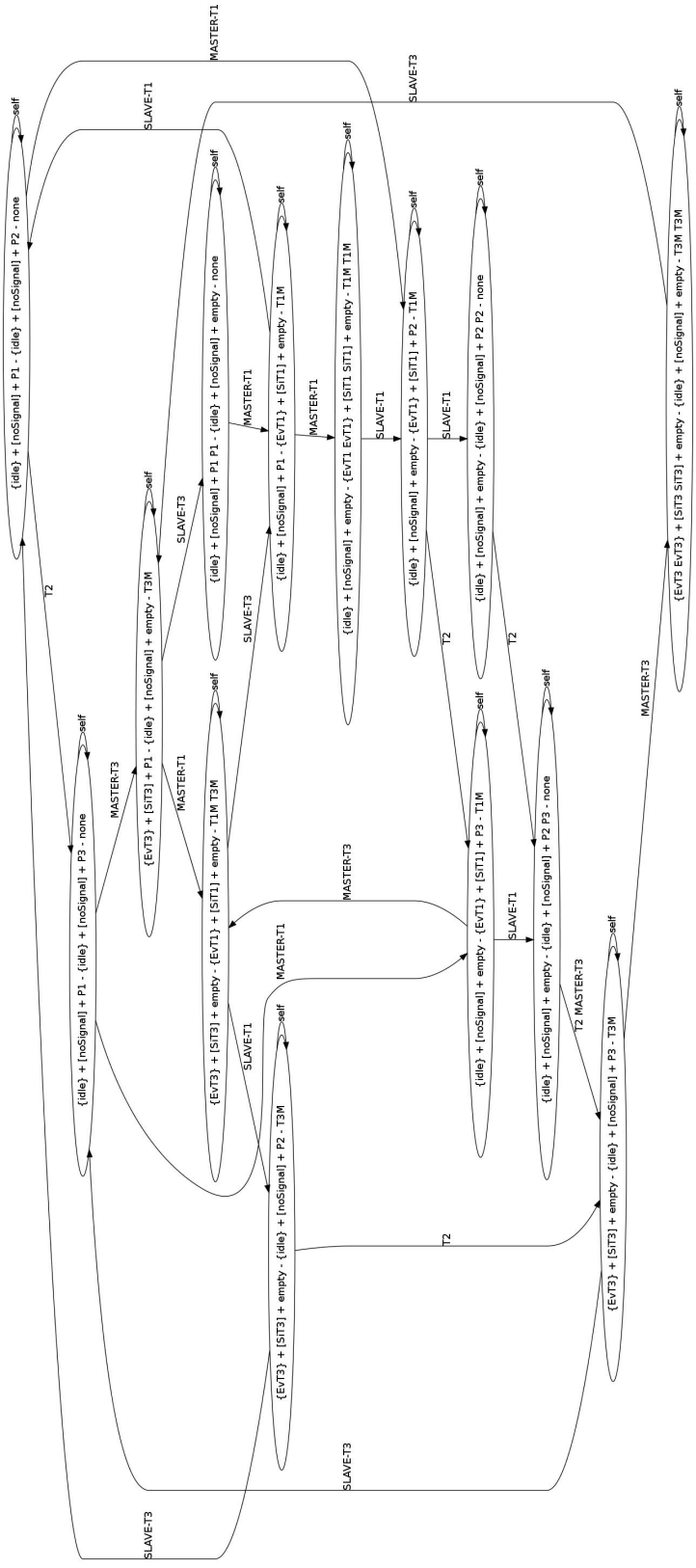

Figura 37: Espaço de estados para globalmente assíncronos e localmente síncronos

observações apontadas.

Código 4.14 em ATL. Especificação do mapeamento lambda envolvendo também redes IOPT como um helper

```
1: helper def: lambda_r2(t : PetriNet!Transition) : IOPT!SynchronySet =
2:   IOPT!SynchronySet.allInstances()->
3:    select(ss | ss.master.name.text = t.name.text+'master')->first();
```

Para a função de especificação de conteúdo da semântica de execução, nomeada por σ, temos quo helper ATL apresentado no Código 4.15 ilustra como lidar com esse procedimento. Para a regra #2 da Splitting como `sigmaMout_r2`, tem-se a representação da semântica de execução de canal síncrono (SynchronySet) `ss`. Define-se uma sequência de transições escravas `slavesSeq` excluindo-se a primeira (linhas 2-3). Assim, compõe-se a transição master `ss.master`, com o operador de composição sequencial `;`, seguido da primeira transição escrava `ss.slave...->first` com composições paralelas + com demais transições escravas em `slavesSeq`, se existirem (linhas 4-7).

Código 4.15 em ATL. Especificação do mapeamento sigma para modelo de saída como um helper

```
1: helper def: sigmaMout_r2(ss : IOPT!SynchronySet) : String =
2:   let slavesSeq : Sequence(IOPT!Transition) =
3:      ss.slaves.excluding(ss.slaves.asSequence()->first())
4:   in ss.master.name.text + ';' + '(' +
5:      ss.slaves.asSequence()->first().name.text +
6:      slavesSeq->iterate(ts; res : String = '' | res + '+' + ts.name.text) +
7:      ')';
```

Assim, se na especificação de um mapeamento λ_r2, envolvendo redes de Petri e redes IOPT, este mapeamento tiver a forma λ_r2: Transition $\rightharpoonup$ SynchronySet, mais os mapeamentos σ, teremos o preenchimento de cada um dos elementos como o par $line_i$ = ($t \in$ Transition, $\lambda\_r2(t) \in$ SynchronySet) desde que t seja mapeável através de λ_r2. O Código 4.16 sumariza a geração do mapeamento `extractTES`, ilustrando o código da regra #2 da transformação Splitting. Como o método de geração é o mesmo empregado na Seção 4.2.5, a estrutura do código é bem similar ao que foi ali gerado. Na linha 2, o código especifica a criação de uma tabela `tes` a partir de dois modelos (`Min` e `Mout`). A regra principal chama-se `Models2Table_r2` (linhas 3-14), tomando uma rede de Petri e uma rede IOPT, e para cada transição de `Min` descobre através de `lambda_r2` se existe um canal síncrono (`SynchronySet`) originado em `Mout`. Para cada caso existente, constrói-se uma linha na tabela através da regra `CreateLine` (linhas 14-19), onde para

a primeira célula dessa linha, a regra `CreateMinCell_r2` (linhas 20-22) cria uma célula correspondente com conteúdo da ação de execução da transição, enquanto que para a segunda célula, a regra `CreateMoutCell_r2` (linhas 23-25) faz com que o conteúdo seja dado através dos operadores categóricos das transições que compõem o canal síncrono.

Código 4.16 em ATL. Sumário do mapeamento extractTES

```
1:  module extractTES;
2:  create tes : TES from Min : PetriNet, Mout : IOPT;
...
3:  rule Models2Table_r2 {
4:   from Min: PetriNet!PetriNet, Mout: IOPT!PetriNet
5:   using { ss : IOPT!SynchronySet = OclUndefined; }
6:   to table: TES!Table
7:   do {
8:    for (t in Min.pages.first().objects.oclIsKindOf(PetriNet!Transition)) {
9:     ss <- thisModule.lambda_r2(t);
10:    if (not ss.oclIsUndefined())
11:     table.lines.append(thisModule.CreateLine_r2(t,ss));
12:   }
13:  }
14: }
...
15: rule CreateLine_r2(t: PetriNet!Transition, ss: IOPT!SynchronySet) {
16:  to line: TES!Line(
17:       minCell <- thisModule.CreateMinCell_r2(t), moutCell <- thisModule.CreateMoutCell_r2(ss)
18:    )
19: }
...
20: rule CreateMinCell_r2(t: PetriNet!Transition) {
21:  to cell: TES!Cell(content <- thisModule.sigmaMin_r2(t))
22: }
...
23: rule CreateMoutCell_r2(ss: IOPT!SynchronySet) {
24:  to cell: TES!Cell(content <- thisModule.sigmaMout_r2(ss)
25: }
```

Assim, para a definição das regras analisadas anteriormente, em especial para a ilustração da regra #2, em caso de haver um mapeamento através de `lambda_r2` de uma transição $T1$ para um canal síncrono $SS1$ entre uma partição mestra e uma escrava, a execução de `extractTES`, através da regra `Models2Table_r2`, produz a TES descrita na Tabela 5. Em caso de haver um número de partições escravas $n > 1$, a TES gerada teria as características apresentadas na Tabela 6.

Finalmente, a especificação que a TES nos provê é útil para a geração de fórmulas que definam

M_{in}	M_{out}
T1	T1(master) ; T1_copy(slave)

Tabela 5: Exemplo de TES com número de escravos n = 1

M_{in}	M_{out}
T1	T1(master) ; (T1_copy(slave)$_1$ + T1_copy(slave)$_2$ + ... + T1_copy(slave)$_n$)

Tabela 6: Exemplo de TES com número de escravos n > 1

a semântica de execução de canais de comunicação gerados nas redes IOPT, a partir de transições em redes de Petri, após o particionamento através da Splitting. Estes canais de comunicação são fechados, ou seja, os modelos particionados podem se comunicar com outros modelos particionados mas não com elementos fora do sistema. As ações de comunicação são executadas em dois passos: (i) a transição *master* envia a mensagem para o canal; e (ii) as transições *slave* consomem em paralelo a mensagem disponível no canal. Essa passagem de mensagem pode assumir diversas semânticas de execução, principalmente quando estivermos no domínio das redes IOPT, como por exemplo, o sincronismo com o consumo em um passo atômico, ou o assincronismo, sendo consumido em algum passo posterior.

Em nosso método de verificação definido, inicialmente temos uma especificação φ que produzirá, através de uma geração textual, duas especificações complementares φ_{in} e φ_{out} aplicáveis a M_{in} e M_{out} respectivamente. Para o domínio do canal de comunicação descrito na TES, formalizou-se, sob a assistência do *Cliente MDA* e da especificação disponível em [Costa, 2010], o seguinte conjunto de propriedades Φ sobre o comportamento de M_{out} a partir de M_{in}, considerando-se o mapeamento que existe através de λ_r_i.

- Considere φ_1 como a seguinte especificação: se a TES relaciona M_{in} e M_{out} através do particionamento de uma transição t, deveremos garantir que o canal gerado sempre consumirá todas as mensagens produzidas. Desta forma, temos φ_{in} = t, com $t \in M_{in}$ representando o disparo de uma transição t pertencente a M_{in} e φ_{out} = [](*t*(master) $\rightarrow$ <>(*t*_copy(slave)1 /\ ... /\ *t*_copy(slave)n). Isto significa que, como no campo M_{out} da TES, teremos a forma $t(master)$ **;** ($t\_copy(slave)_1 \oplus t\_copy(slave)_2 \oplus$ ... $t\_copy(slave)_n$). O operador de composição sequencial (**;**) foi traduzido para o operador proposicional implica ($\rightarrow$) seguido do operador futuramente (<>). Já o operador de composição paralela (**+**) exige uma ocorrência simultânea de disparo nas transições escravas, exigindo que se verifique a ocorrência

de todas essas ações em um estado futuro.

- Considere φ_2 como a seguinte especificação: se a TES relaciona M_{in} e M_{out} através do particionamento de uma transição t, deveremos garantir que o canal gerado não poderá realizar uma operação de escrita e uma de leitura ao mesmo tempo, exigindo pelo menos dois passos. Desta forma, temos φ_{in} = t, com $t \in M_{in}$ representando o disparo de uma transição t pertencente a M_{in} e φ_{out} = [](*t*(master) $\rightarrow$ O(*t*(master)Msg U *t*_copy(slave)1 /\ ... /\ *t*_copy(slave)n). Isto significa dizer que a transição *t*(master) executa uma ação e no próximo estado gera uma mensagem (*t*(master)Msg) que deve permanecer no canal até que todos os escravos sejam disparados.

- Considere φ_3 como a seguinte especificação: se a TES relaciona M_{in} e M_{out} através do particionamento de uma transição t, deveremos garantir que o canal gerado deverá preservar a ordem de consumo para as mensagens geradas. Isto quer dizer que, se uma mensagem m_0 for produzida antes de uma mensagem m_1, deveremos ter m_0 sendo consumida antes de m_1. Desta forma, temos φ_{in} = t /\ O(t), com $t \in M_{in}$ representando o disparo sequencial duplo de uma transição t pertencente a M_{in} e φ_{out} = []((t(master) /\ t(master)$Msg0$) /\ ¬(t(master) /\ t(master)$Msg1$) /\ <> (t(master) /\ t(master)$Msg1$) $\rightarrow$ <> (t_copy(slave) /\ t(master)$Msg1$) /\ (t_copy(slave))). Isto significa dizer que, deve haver um controle caso inicialmente seja gerada uma mensagem t(master)$Msg0$ antes de uma mensagem t(master)$Msg1$ para que essa mensagem t(master)$Msg0$ seja consumida antes de t(master)$Msg1$.

A Tabela 7 sumariza as fórmulas geradas a partir da TES gerada como conjuntos Φ_{in} e Φ_{out} a serem utilizadas na verificação de propriedades do canal de comunicação originado pela Splitting.

Propriedade	φ_{in}	φ_{out}
φ_1	T1	[](T1(master) -> <>(T1_copy(slave))
φ_2	T1	[](T1(master) -> O(T1(master)Msg U T1_copy(slave))
φ_3	T1 /\ O(T1)	[](T1(master) /\ T1(master)Msg0) /\ ¬(T1(master) /\ T1(master)Msg1) /\ <> (T1(master) /\ T1(master)Msg1) -> <> (T1_copy(slave) /\ T1(master)Msg1) /\ (T1_copy(slave)))

Tabela 7: Exemplo de TES com número de escravos n = 1

4.4 Recapitulação e Considerações

Este capítulo apresentou a principal instanciação existente da MDA-Veritas, que serve para a verificação de equivalência de transformações envolvendo modelos em redes de Petri e redes IOPT de sistemas concorrentes. Após o preenchimento de cada item da arquitetura, espera-se, através de sua aplicação, que a instanciação apresente bastante utilidade para o domínio.

Para uma descrição complementar a esta instanciação da MDA-Veritas para sistemas concorrentes, o leitor pode consultar o trabalho [Barbosa et al., 2009a]. Maiores detalhes do novo projeto sintático e semântico da linguagem IOPT, que foi um dos frutos originados por colaboração com nosso projeto, podem ser obtidos diretamente em [Moutinho et al., 2010].

5 Verificação de Transformações na MDA-Veritas Instanciada

Este capítulo descreve aplicações da instanciação da MDA-Veritas para sistemas concorrentes apresentada no capítulo anterior. Ilustramos a aplicação da MDA-Veritas no contexto do projeto *FORDESIGN*. Este projeto explora vários modelos distintos de computação para projeto de sistemas embarcados, e exerce o papel do *Cliente MDA* no processo de uso da MDA-Veritas. Faz-se uso de redes de Petri como uma linguagem de especificação independente de plataforma para modelar sistemas e componentes concorrentes, e de sua linguagem proposta, as IOPTs, para detalhes específicos de plataforma, nos quais são verificados e implementados em hardware ou software usando técnicas de co-projeto [Gomes and Costa, 2006]. Analisamos e avaliamos a transformação MDA *Splitting* representando a operação de fragmentação de rede, que permite decompor um modelo em rede de Petri em sub-modelos de redes de Petri usando canais de comunicação síncronos [Gomes and Costa, 2007]. As provas de corretude apresentadas das transformações para particionamento de modelos e identificação de sub-modelos buscam dar suporte ao ciclo de desenvolvimento completo dos sistemas, da especificação até a implementação.

Detalhamos as verificações de equivalência para casos dessa transformação nas seções 5.1, 5.2 e 5.3, como PIM-para-PIM, PSM-para-PSM e PIM-para-PSM, respectivamente. Finalmente, a Seção 6.4 faz as considerações finais sobre o capítulo e indica material complementar.

5.1 Aplicação de uma Transformação PIM-para-PIM

Nesta seção, analisamos a aplicação da técnica de verificação de semântica em transformações MDA para provar equivalência de modelos independentes de plataforma (PIMs). Isto corresponde à primeira ilustração de uso prático da instanciação fornecida para a MDA-Veritas. Esta abordagem complementa-se com técnicas e ferramentas específicas para lidar com a essência do paradigma concorrente.

5.1.1 Definição dos Modelos

Aplicamos a verificação de preservação de propriedades nos modelos envolvidos na transformação Splitting. Para propósitos ilustrativos, abordamos um modelo que descreve um controlador de estacionamento como ilustrado na Figura 38. O software controlador deve estar presente nas duas máquinas que controlam os portões de entrada e de saída do estacionamento. A rede de Petri que

modela o sistema controlador como um todo é mostrada na Figura 39. A entrada e a saída são modeladas usando-se, respectivamente, as partes esquerda e direita do modelo. O controlador de estacionamento possui uma entrada e uma saída (lugares `EntranceFree` e `ExitFree`), e alguns lugares de estacionar (lugar `FreePlaces` com quatro marcas). Estes lugares modelam o estado do controlador de estacionamento. Desta forma, quando um carro chega ao portão, é disparada a transições `Arrive+`, que coloca uma marca no lugar `WaitingTicket`. Ao ocorrer a entrada desse carro no estacionamento, é disparada a transição `Enter`, além de retirar a marca de `WaitingTicket`, abre-se o portão (inserindo marca em `GateOpen`), insere-se o carro dentro da zona (inserindo marca em `CarInsideZone`) e ocupa-se um lugar do estacionamento (retirando marca de `FreePlaces`). Já para a saída do carro, com o disparo da transição `Exit`, além de retirar a marca de `CarInsideZone`, devolve-se uma marca a `FreePlaces`, coloca-se uma marca em `CarInsideZone` e retira-se uma marca de `WaitingToPay`, que deve ter sido preenchido anteriormente através do disparo da transição `Leave-`, que significa a saída do carro do estacionamento.

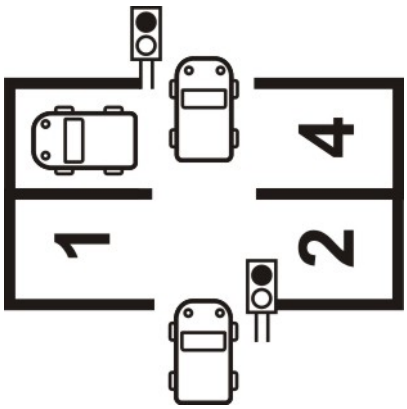

Figura 38: Um controlador de estacionamento simples

A transformação Splitting é aplicada ao modelo como uma transformação ATL. O modelo particionado gerado é mostrado na Figura 40. Podemos observar que a transição `Enter` foi escolhida como ponto principal de corte, implicando na fragmentação dos componentes.

5.1.2 Definição das Propriedades a Serem Observadas nos Modelos

Para este caso, de acordo com o contexto do projeto em questão, ambos os modelos são equivalentes se duas condições principais forem satisfeitas: (i) se as propriedades de *ausência de deadlock* e/ou *vivacidade* ocorrerem em M_{in}, então isto também deverá ser válido para M_{out}; e (ii) ambos

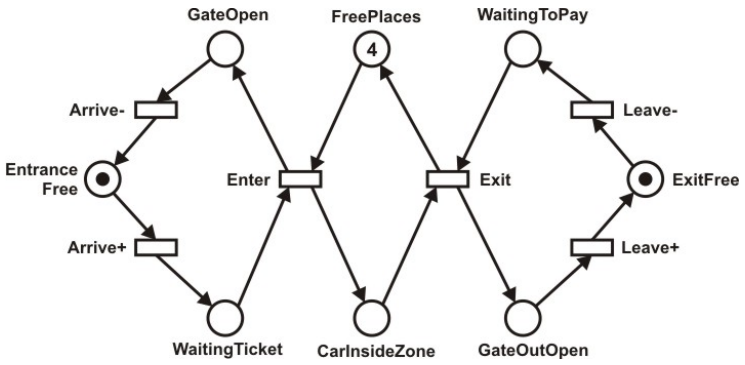

Figura 39: A rede que modela o controlador de estacionamento

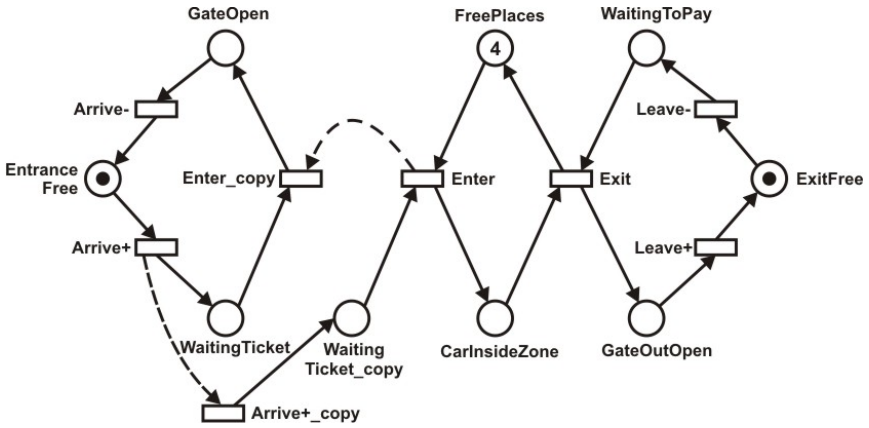

Figura 40: O modelo de saída fragmentado do controlador de estacionamento

modelos preservarem a mesma ordem parcial de ocorrência de eventos, ou seja, para uma dada seqüência de transições, eles produzirem caminhos de marcações com a mesma ordem de disparo de transições e ambos alcançarem o mesmo estado, ou marcação. Neste sentido, executamos a verificação das propriedades enumeradas para cada modelo como verificação de equivalência através de verificação de modelos complementando o uso do *Verificador Formal* com esta quarta técnica.

5.1.3 Emprego da MDA-Veritas

Aqui, ilustramos o uso da MDA-Veritas, também chamado de atividade *Empregar Técnicas de Análise*, atribuída ao ator *Cliente MDA*. Devido às técnicas escolhidas na instanciação da MDA-Veritas, espera-se que essa verificação seja praticamente automatizada em sua totalidade, requisitando apenas a especificação das propriedades a serem verificadas. Assim, empregaremos todo o trabalho desenvolvido pelo cliente *Provedor da Solução*, durante o preenchimento da instanciação através do emprego do resultado das cinco atividades.

Emprego da Atividade i. *Prover regras de boa formação a partir da sintaxe de M_{in} e M_{out}.* Submetemos os modelos envolvidos na transformação (M_{in} e M_{out}) às regras OCL determinadas para este domínio, que podem ser melhor detalhadas em [Moutinho et al., 2011] e que não fazem parte da questão principal deste trabalho. Com o resultado de satisfação, estamos aptos a seguir com o emprego das outras técnicas, pois estamos lidando agora com modelos bem-formados.

Emprego da Atividade ii. *Instanciação através do metamodelo semântico.* Este passo visou garantir que o *metamodelo semântico* provido para a Categoria Petri esteja sendo utilizado em nosso framework. Ele foi construído no formato Ecore [Budinsky et al., 2003], permitindo sua manipulação pelo framework ATL-DT. Este emprego deste metamodelo é automatizado, sendo utilizado pelas equações semânticas logo a seguir.

Emprego da Atividade iii. *Extração através das equações semânticas.* A aplicação das *equações semânticas* em ambos os modelos produz os *modelos semânticos*. Essas equações trabalham a partir do *metamodelo semântico* Categoria Petri, garantindo uma generalização de funcionamento para o esquema mais tradicional de definição das redes de Petri. O modelo semântico representa conceitos para este domínio como multiconjuntos e regras que especificam comportamento das transições. Este *modelo semântico*, gerado no formato Ecore para ser usado de uma maneira genérica no framework MDA, passou a ser denominado, mais recentemente, como *Modelo Semântico Independente de Plataforma* [Barbosa et al., 2009a]. Cada item apresentado é muito importante para o próximo passo: a geração de código como *Modelo Semântico Específico de Plataforma*, atrelado às redes IOPT, para propósitos de verificação.

Emprego da Atividade iv. *Uso da especificação do processamento dos modelos semânticos.* Para a semântica dinâmica, o disparo de uma transição foi definido como uma regra de reescrita, conforme as linhas 9-14 do *modelo semântico extraído* para M_{in} conforme apresentado no Código 5.1. Isto signifca que utiliza-se alguma plataforma operacional para permitir a execução automática e o processamento dessas regras de computação dos modelos. Usamos esta definição como a representação concreta possível de comportamento para guiar as análises no *Verificador Formal*. Esta teoria é gerada através de mapeamentos ATL de modelo para modelo e de mapeamentos MOFScript modelo para texto. Isto corresponde às equações semânticas sendo empregadas de modelos redes de Petri para modelos Maude.

Código 5.1 em Maude. Representação do modelo de entrada

```
1: mod INPUT-PARKING-LOT is
2:  sorts Place Marking .
3:  subsort Place < Marking .
4:  ops EntFree GtOpen FreePcs WaitnPay ExitFree
5:  GtOutOpn CarInZone WaitnTkt : -> Place .
6:  op empty : -> Marking .
7:  op __ : Marking Marking
8:  -> Marking [assoc comm id: empty] .
9:  rl[Arrive-] : GtOpen => EntFree .
10: rl[Arrive+] : EntFree => WaitnTkt .
11: rl[Enter] : FreePcs WaitnTkt => GtOpen CarInZone .
12: rl[Exit] : CarInZone WaitnPay => FreePcs GtOutOpn .
13: rl[Leave-] : GtOutOpn => ExitFree .
14: rl[Leave+] : ExitFree => WaitnPay .
15:endm
```

Partindo dos *modelos semânticos* extraídos pelas *equações semânticas* e denominados de *Modelo Semântico Independente de Plataforma*, temos que eles devem ser novamente extraídos automaticamente para uma plataforma específica de verificação, devido ao fato de que nem as ferramentas Eclipse/Ecore e nem o framework MDA atual permitirem o raciocínio sobre a semântica operacional dos modelos. O sistema escolhido para o processamento dessas representações foi o sistema de reescrita Maude, que é capaz de representar nossa formalização de redes de Petri usando *teoria das categorias* para lógica de reescrita. Como ilustração, o Código 5.2 apresenta o denominado *Modelo Semântico Específico de Plataforma*, que é a representação em Maude de M_{in}. Todos os lugares empregados na rede são operações (linhas 4-5), uma marcação é definida como vazia ou como concatenação de outras marcações (linhas 6-7), e as transições existentes são geradas como regras (linhas 9-14).

O Código 5.2 provê os componentes particionados para M_{out}. As transições `Arrive+` e `Enter` carregam as mensagens para serem trocadas entre as redes (linha 12), e suas regras (linhas 18-23) mudam concorrentemente as marcações de ambas redes de acordo com a especificação. De maneira complementar, as transições restantes são representadas como ações internas (linhas 16-17 e 25-30) devido ao fato de que seu disparo não altera marcações fora do escopo da rede que as possui.

Código 5.2 em Maude. Representação do modelo de saída

```
1: mod OUTPUT-PARKING-LOT is
2:  inc CONFIGURATION .
3:  sorts Place Marking .
4:  subsort Place < Marking .
5:  ops EntFree GtOpen FreePcs WaitnPay ExitFree
6:  GtOutOpn CarInZone WaitnTkt WaitnTktM : -> Place .
7:  op empty : -> Marking .
8:  op __ : Marking Marking
9:   -> Marking [assoc comm id: empty] .
10: op IOPT : -> Cid [ctor] .
11: op m :_ : Marking -> Attribute [ctor gather (&)] .
12: vars Comp1 Comp2 : Oid .
13: ops Arrive+;Arrive+_copy(Comp1,Comp2)  Enter;Enter_copy(Comp1,Comp2) : Oid Oid -> Msg [ctor] .
14: var C : Configuration .
15: var Any : Marking .
16: rl[Arrive-] : <Comp1:IOPT | m:GtOpen> =>
17: <Comp1:IOPT | m:EntFree> .
18: rl[Arrive+;Arrive+_copy] : Arrive+;Arrive+_copy(Comp1,Comp2)
19:  <Comp1:IOPT | m:EntFree> <Comp2:IOPT | m:Any>
20:  => < Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT | m:Any WaitnTktM> .
21: rl[Enter;Enter_copy] : Enter;Enter_copy(Comp1,Comp2)
22:  <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT | m:Any WaitnTktM FreePcs>
23:  => <Comp1:IOPT | m:GtOpen> <Comp2:IOPT | m:Any CarInZone> .
25: rl[Exit] : <Comp2:IOPT | m:CarInZone WaitnPay Any> =>
26:  <Comp2:IOPT | m:FreePcs GtOutOpn Any> .
27: rl[Leave-] : <Comp2:IOPT | m:Any GtOutOpn> =>
28:  <Comp2:IOPT | m:Any ExitFree> .
29: rl[Leave+] : <Comp2:IOPT | m:Any ExitFree> =>
30:  <Comp2:IOPT | m:Any WaitnPay> .
31:endm
```

Emprego da Atividade v. *Aplicação do Verificador Formal para expressar a equivalência dos modelos.* A seguir, analisamos, em detalhes, todos os passos necessários para verificar equivalência semântica entre os modelos gerados para o exemplo do controlador de estacionamento considerando as técnicas para o *Verificador Formal*. Começamos reusando M_{in} e M_{out} após a execução da transformação Splitting, produzindo a *tabela de equivalência* apresentada na Tabela 8. Essa tabela será reusada posteriormente para garantir a equivalência entre as transições durante a verificação de equivalência.

Especificamos os estados iniciais de ambos os *modelos semânticos* para que propriedades de equivalência semântica a serem verificadas possam também serem especificadas. Temos que

```
eq initial = (EntranceFree ExitFree FreePcs FreePcs FreePcs FreePcs)
```

Modelo de Entrada	Modelo de Saída
Arrive-	Arrive-
Arrive+	Arrive+;Arrive+_copy
Enter	Enter;Enter_copy
Exit	Exit
Leave-	Leave-
Leave+	Leave+

Tabela 8: Tabela de Equivalência para o controlador de estacionamento

seja o estado inicial de M_{in}, significando que a rede será simulada possuindo uma marca no lugar *Entrance Free* (`EntFree`), uma marca no lugar *Exit Free* (`ExitFree`) e quatro fichas no lugar *Free Places* (`FreePcs`), como atesta a Figura 39. Também temos que

```
eq initial = Enter;Enter_copy(Comp1, Comp2)
  <Comp2:IOPT | m:(FreePcs FreePcs FreePcs FreePcs ExitFree)>
  Arrive+;Arrive+_copy(Comp1, Comp2) <Comp1:IOPT | m:EntFree >
```

seja o estado inicial de M_{out}, significando que a rede será simulada possuindo uma marca no lugar *Entrance Free* (`EntFree`) para o componente 1 (`Comp1`), uma marca no lugar *Exit Free* (`ExitFree`) para o componente 2 (`Comp2`) e quatro marcas no lugar *Free Places* (`FreePcs`) para o componente 2 (`Comp2`), como atesta a Figura 40. Além do mais, M_{out} requer a especificação dos canais `Enter;Enter_copy` e `Arrive+;Arrive+_copy` para a comunicação entre `Comp1` e `Comp2`.

Ausência de Deadlock e Vivacidade. Para o *modelo semântico* de M_{in}, utilizamos o comando na linha 1 da Computação 5.1, começando da marcação inicial. Após isto, a última linha exibe `No solution.`, que significa que a especificação gerada para M_{in} é *livre de deadlock*.

Computação 5.1 em Maude. Verificação de ausência de deadlock para o modelo de entrada

```
1: search in INPUT-PARKING-LOT : initial =>! Any:Marking .
2: No solution.
3: states: 54 rewrites: 102 in 0ms cpu (~rew/sec)
```

A mesma propriedade deve ser satisfeita para o *modelo semântico* de M_{out}. Ao aplicar o comando nas linhas 1-4 da Computação 5.2, perguntamos se o sistema particionado alcança um estado que não tenha sucessores. Ao final, nenhum estado sem sucessor é encontrado, representando o mesmo tipo de comportamento verificado para M_{in}.

Computação 5.2 em Maude. Verificação de ausência de deadlock para o modelo de saída

```
1: search in OUTPUT-PARKING-LOT : Enter;Enter_copy(Comp1, Comp2)
2: <Comp2:IOPT | m:(FreePcs FreePcs FreePcs FreePcs ExitFree)>
3: Arrive+;Arrive+_copy(Comp1, Comp2))
4: <Comp1:IOPT | m:EntFree> =>! C:Configuration .
5: No solution.
6: states: 35 rewrites: 55 in 0ms cpu (~rew/sec)
```

Complementar à propriedade de *ausência de deadlock*, *vivacidade* garante que em ambos modelos, globalmente todos os estados possuem marcações com pelo menos uma transição habilitada. A propriedade `enabled` foi especificada novamente em um módulo separado usando Maude. Ao observar as Computação 5.3 e Computação 5.4, ambos modelos apresentam a mesma saída quando submetidos à verificação dessa propriedade.

Computação 5.3 em Maude. Saída da propriedade enabled para o modelo de entrada

```
1: reduce in INPUT-PARKING-LOT : modelCheck(initial, []enabled) .
2: rewrites: 8 in 0ms cpu (0ms real) (~ rew/sec)
3: result Bool: true
```

Computação 5.4 em Maude. Saída da propriedade enabled para o modelo de saída

```
1: reduce in OUTPUT-PARKING-LOT :
2: modelCheck(initial, []enabled) .
3: rewrites: 26 in 0ms cpu (0ms real) (~ rew/sec)
4: result Bool: true
```

Preservando a Ordem dos Eventos. A verificação foi executada automaticamente para todos os caminhos de comportamento gerados dos modelos. No intuito de mostrar que um número de transições pode ser disparado em uma dada seqüência, como exemplo, apresentamos o disparo das transições `Enter,Arrive-, Arrive+, Leave+, Exit`. A linhas 1, 2 e 3 nos Computação 5.5 e Computação 5.6 ilustram como isto é especificado em LTL para a ferramenta Maude model-checker. A mesma seqüência é disparada, de acordo com a *tabela de equivalência* e ambos os modelos alcançam o mesmo estado. Por exemplo, o estado alcançado está na linha 13 para `INPUT-PARKING-LOT` (modelo semântico de M_{in}) e nas linhas 16-17 para `OUTPUT-PARKING-LOT` (modelo semântico de M_{out}). Então, concluiu-se positivamente sobre a equivalência semântica entre os modelos, analisada de acordo com os resultados providos pelos contra-exemplos.

Computação 5.5 em Maude. Verificação da ordem dos eventos para o modelo de entrada

```
1: reduce in INPUT-PARKING-LOT: modelCheck(initial,
2: ~ <> (Enter /\ O(Arrive- /\ O (Arrive+
3: /\ O (Leave+ /\ O Exit))))) .
3: rewrites: 255 in 5ms cpu (48000 rew/sec)
4: result ModelCheckResult:
5: {FreePcs FreePcs FreePcs FreePcs ExitFree WaitnTkt,'Enter}
7: {GtOpen FreePcs FreePcs FreePcs ExitFree CarInZone,'Arrive-}
9: {EntFree FreePcs FreePcs FreePcs ExitFree CarInZone,'Arrive+}
11:{FreePcs FreePcs FreePcs ExitFree CarInZone WaitnTkt,'Leave+}
13:{FreePcs FreePcs FreePcs WaitnPay CarInZone WaitnTkt,'Exit}
```

Computação 5.6 em Maude. Verificação da ordem dos eventos para o modelo de saída

```
1: reduce in OUTPUT-PARKING-LOT :
2: modelCheck(initial,
3: ~ <> (Enter;Enter_copy /\ O(Arrive- /\ O (Arrive+;Arrive+_copy /\ O (Leave+ /\ O Exit))))) .
4: rewrites: 92 in 3ms cpu (47000 rew/sec)
5: result ModelCheckResult:
6: {Arrive+;Arrive+_copy(Comp1, Comp2) Enter;Enter_copy(Comp1, Comp2) Enter;Enter_copy(Comp1,Comp2)
7: <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT | m:(FreePcs FreePcs
8: FreePcs FreePcs ExitFree WaitnTktM)>,'Enter;Enter_copy}
9: {Arrive+;Arrive+_copy(Comp1, Comp2) Enter;Enter_copy(Comp1, Comp2) <Comp1:IOPT | m:GtOpen>
10:<Comp2:IOPT | m:(FreePcs FreePcs FreePcs ExitFree CarInZone)>,'Arrive-}
11:{Arrive+;Arrive+_copy(Comp1, Comp2) Enter;Enter_copy(Comp1, Comp2)
12:<Comp1:IOPT | m:EntFree> <Comp2:IOPT | m:
13:(FreePcs FreePcs FreePcs ExitFree CarInZone)>,'Arrive+;Arrive+_copy}
14:{Enter;Enter_copy(Comp1, Comp2) <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT |
15:m:(FreePcs FreePcs FreePcs ExitFree CarInZone WaitnTktM)>,'Leave+}
16:{Enter;Enter_copy(Comp1, Comp2) <Comp1:IOPT | m:WaitnTkt> <Comp2:IOPT |
17:m:(FreePcs FreePcs FreePcs WaitnPay CarInZone WaitnTktM)>,'Exit}
```

5.2 Aplicação de uma Transformação PSM-para-PSM

Nesta seção, analisamos a nossa abordagem para verificar equivalência entre modelos específicos de plataforma. A transformação analisada continua sendo a *Splitting*, porém, fazendo uso de modelos descritos em IOPT, que agregam conceitos específicos das plataformas para sistemas embarcados denominadas por PICs (Programmable Interface Controller) [Mazidi et al., 2005]. Analisamos o processo de definição e de extração dos *modelos semânticos* para esse domínio, os espaços de estados provenientes de cada semântica de execução e buscamos prover a verificação desejada pelo *cliente MDA*, que é o provedor da transformação.

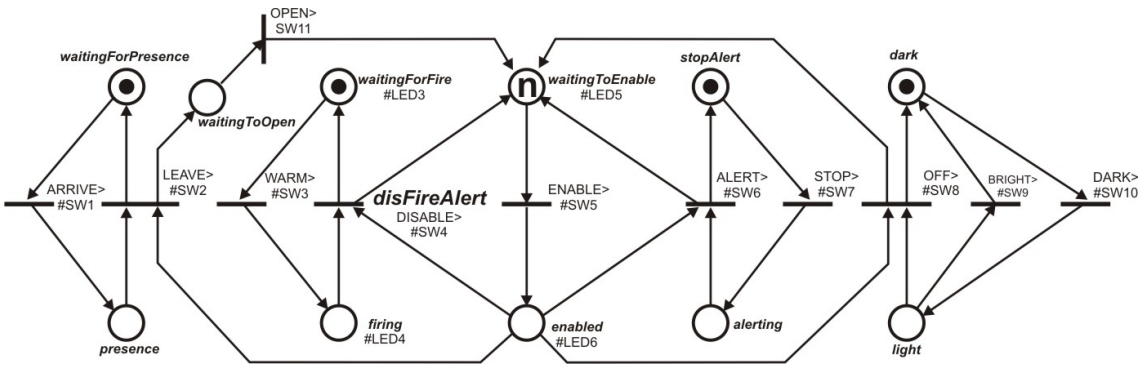

Figura 41: Modelo domótico genérico

5.2.1 Definição dos Modelos

Tomamos um cenário em que a transformação *Splitting* também decompõe modelos IOPTs e utiliza modelos de comunicação específicos de acordo com a plataforma a ser considerada. A Figura 41 apresenta um exemplo de uma rede IOPT utilizada para controle de diversas funções em um condomínio residencial segundo a tecnologia de domótica. O modelo reusa a estrutura básica das redes de Petri e possui características das IOPTs, tais como eventos de entrada (ARRIVE, LEAVE, OPEN, WARM, DISABLE, ENABLE, ALERT, STOP, OFF, BRIGHT, DARK) e sinais de entrada (SW1, SW2, SW3, SW4, SW5, SW6, SW7, SW8, SW9, SW10 e SW11). Contudo, conforme mencionado anteriormente, as características mais importantes não podem ser vistas sintaticamente, devido estarem no domínio semântico. Exemplos disso, são as semânticas de execução específicas a serem adotadas pelos modelos, as prioridades entre transições conflitantes ou a ocorrência no ambiente dos fatores que influenciam a rede não-autônoma.

O sistema deve gerenciar energia e portões de condomínio, emitir mensagens de alerta e proteger de assaltantes e incêndios. O modelo pode ser entendido através de regiões de nós de redes de Petri. Na Figura 41, mais à esquerda, através das transições com eventos ARRIVE, LEAVE e OPEN temos um módulo detector de presenças estranhas no condomínio. Após a detecção e tudo ser resolvido, o portão deve ser aberto para a saída. À esquerda do centro, temos um componente de detecção de incêncio, com os eventos WARM e DISABLE. Ao centro, temos o acionador principal, chamado pelo evento ENABLE. À direita do centro, temos o módulo de alerta contra fatos inesperados, possuindo os eventos ALERT e STOP. Finalmente, temos mais à direita o módulo de controle de luminosidade com os eventos OFF, BRIGHT e DARK. Dependendo da disposição geográfica do condomínio, do número de microcontroladores disponíveis e outros requisitos mais,

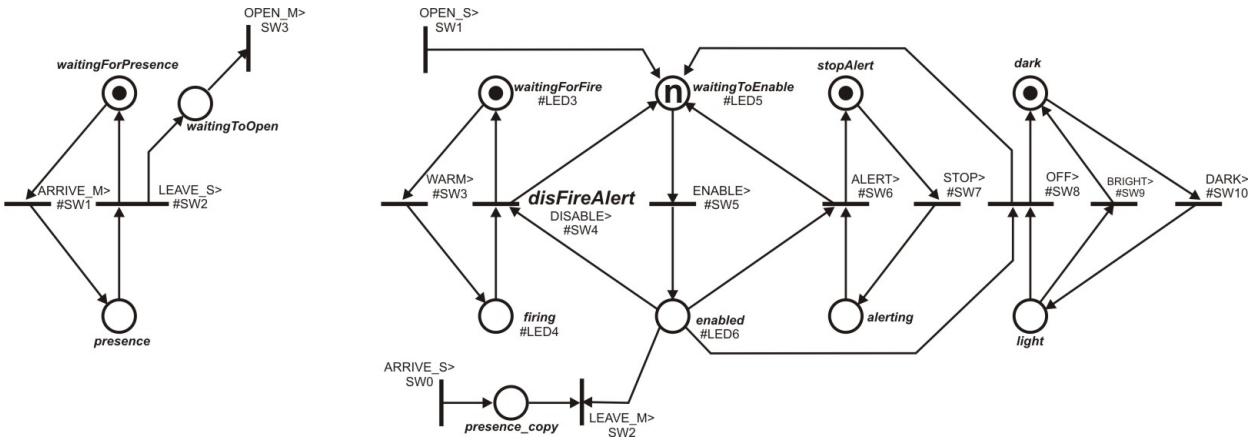

Figura 42: Modelo domótico particionado em dois módulos

a transformação *Splitting* poderia dividir esse modelo em até cinco componentes.

Por simplicidade, e por ser um caso empregado na prática por uma aplicação industrial parceira deste trabalho, avaliemos aqui a aplicação da *Splitting* para obtermos dois componentes independentes: o detector de presenças estranhas (Módulo 1) e o restante do sistema (Módulo 2). Isto se deve pelo limite de hardware disponível, por haver disponível no momento apenas comunicação através do padrão RS-232, pela proteção contra assaltos ser requisito essencial e estar disponível em uma empresa de segurança contratada. A Figura 42 apresenta esse sistema particionado em dois módulos de acordo com a transformação *Splitting*.

5.2.2 Definição das Propriedades a Serem Observadas nos Modelos

Diversas propriedades foram solicitadas pelo *Cliente MDA* para verificação da equivalência entre os modelos envolvidos na *Splitting*. Esta equivalência também é garantida através de model checking. Esta técnica permite reusar a mesma especificação de propriedades a ambos os modelos.

Tomamos como exemplo uma propriedade qualquer de alcançabilidade a ser verificada. Ela beneficia-se do fato de reusarmos o comando *search* do Maude, a todos os modelos. As demais propriedades a serem verificadas seguem uma derivação automática de acordo com os padrões de especificação de propriedades para LTL. O primeiro exemplo a ser abordado é a verificação do padrão de propriedade de *Existência* entre eventos. Assumindo que uma propriedade P torna-se verdade, queremos saber se P ocorre após Q. Utilizando-se P = ALERT e Q = ENABLE, queremos saber se nos dois modelos, ao ocorrer um estado de alerta, teremos uma habilitação do acionador permitindo a funcionalidade normal de todos os dispositivos da residência. Esperamos obter res-

postas negativas à essa verificação.

Por fim, tomamos em detalhes a verificação sob um padrão de especificação por completo. O padrão escolhido foi o de *Precedência*. Esse padrão busca especificar informações sobre variações dado o cenário de que uma determinada propriedade S precede uma propriedade P. Interessou ao *cliente MDA* considerar S = ENABLE, P = ALERT, Q = OPEN e R = STOP. Esse padrão pode ser especificado a qualquer modelo, desde que um conjunto de propriedades básicas seja enumerado como ocorre aqui. A partir desse cenário, cinco possíveis consultas são possíveis de acordo com a tabela que especifica este padrão no Capítulo 2.4.5. Elas estão enumeradas na Tabela 9.

Propriedade	**Fórmula**
Sempre	!P W S
Antes de R	<>R -> (!P U (S or R))
Após Q	[]!Q or <>(Q and (!P W S))
Entre Q e R	[](Q and !R and <>R) -> (!P U (S or R)))
Após Q até R	[](Q and !R -> (!P W (S or R)))

Tabela 9: Padrões para investigação da precedência entre propriedades

5.2.3 Emprego da MDA-Veritas

Emprego da Atividade i.. *Validação através das regras de boa formação.* Submetemos os modelos envolvidos na transformação (M_{in} e M_{out}) às regras OCL determinadas para este domínio. Após a instanciação dos modelos IOPT de acordo com o metamodelo [Moutinho et al., 2010], reusamos definições em OCL também providas por esse nosso trabalho. Os códigos 5.3, 5.4 e 5.5 ilustram alguns exemplos de verificação da semântica estática.

Código 5.3 em Maude. Fragmento de semântica estática para verificação de arco de teste

```
1: Context TestArc inv SourcePlaceTargetTransition:
2:  self.source.oclIsKindOf(PlaceNode) and self.target.oclIsKindOf(TransitionNode)
```

Código 5.4 em Maude. Fragmento de semântica estática para limites de uma marcação

```
1: Context Bound inv:
2:  self.text <= self.ownerPlace.initialMarking.text
```

Código 5.5 em Maude. Fragmento de semântica estática para diferenciação de transições mestras e escravas

```
1:  Context SynchronySet inv MasterDiffSlaves:
2:   self.slaves->forAll(t|not(t=self.master))
```

Emprego da Atividade ii. *Instanciação através do metamodelo semântico.* O principal uso do *metamodelo semântico* se dá na atividade logo a seguir das equações semânticas. Fazemos forte uso do metamodelo da solução Maude para a geração automatizada das teorias que representam *modelos semânticos*.

Emprego da Atividade iii. *Extração através das equações semânticas.* Para extrair os modelos semânticos, necessitamos de *equações semânticas* descritas como transformações de IOPT para Maude. A especificação Maude é extraída de uma forma completamente automática tendo as *equações semânticas* como transformações MDA para a semântica de execução determinada. Dada a apresentação do processo já feita em seções anteriores, vamos direto ao artefato transformação conforme sua descrição inicia-se no Código 5.6. Essas equações são genéricas para todo modelo definido de acordo com o metamodelo IOPT, porém com domínio específico nas construções Maude em lógica de reescrita.

Código 5.6 em Maude. Regra principal da transformação de IOPT

```
1: rule IOPT2Spec {
2:  from s : iopt!PetriNet
3:  to   t : Maude!MaudeSpec (els <- sm),
4:       sm : Maude!SModule (
5:        name <- s.name.text, els <- s1, els <- s2, els <- s3, els <- s4,
6:        els <- op1, els <- op2, els <- op3, els <- op4, els <- op5,
7:        els <- op6, els <- op7, els <- op8, els <- op9
8:       ),
9:       s1 : Maude!Sort (name <- 'SignalSet'), s2 : Maude!Sort (name <- 'EventSet'),
10:      s3 : Maude!Sort (name <- 'Marking'), s4 : Maude!Sort (name <- 'IOPT'),
11:      op1 : Maude!Operation (name <- 'empty'), op2 : Maude!Operation (name <- 'noSignal'),
12:      op3 : Maude!Operation (name<-'idle'), op4 : Maude!Operation(name<-'noState', coarity<-s4),
13:      op5 : Maude!Operation (name <- '__', atts <- 'assoc', atts<-'comm', atts<-'id: empty'),
14:      op6 : Maude!Operation (name <- self.op1name, atts <- 'ctor', coarity <- s2),
15:      op7 : Maude!Operation (name <- self.op2name, atts <- 'ctor', coarity <- s1),
16:      op8 : Maude!Operation (name <- self.op3name, coarity <- s3),
17:      op9 : Maude!Operation (name <- '_+_+_', arity<-s1, arity <- s2, arity<-s3, coarity<-s4)
18:}
```

Basicamente, transformamos um elemento da classe sintática `PetriNet` em um elemento da classe semântica `MaudeSpec`, onde criamos um `SModule` (módulo de estados) e o preenchemos

com o atributo `name` e preenchemos seus elementos `els` com elementos das redes de Petri. Assim, preencheremos os tipos (`Sort`) e as operações (`Operation`) assim como no template apresentado no Código 4.15. Para cada tipo de *modelo semântico* estabelecido, teremos uma transformação representando uma *equação semântica* específica. A partir daí, temos mapeamentos para a sintaxe concreta Maude.

Emprego da Atividade iv. *Uso da especificação do processamento dos modelos semânticos.* A semântica dinâmica foi definida logo a seguir, para cada uma das possibilidades de execução e comunicação nas plataformas existentes. As regras, ou arcos, que permitem a mudança entre os estados nos grafos de espaços de estados apresentados constituiram a implementação dessa atividade. Assim, poderemos guiar as análises no *Verificador Formal*.

A Figura 43 apresenta a estrutura genérica de *modelos semânticos* que serão gerados para redes IOPT. Este tipo de diagrama é muito utilizado no Maude Manual [Clavel et al., 2011] para mostrar a estrutura de inclusão dos módulos e teorias para uma determinada solução. Haverá instanciação disso para M_{in} e M_{out} e serão explicadas ao longo desta seção. Apresentamos os diversos módulos Maude gerados e utilizados, além das diversas inclusões desses módulos. Ao nível mais baixo, temos o reuso dos módulos pré-definidos CONFIGURATION, que define a parte denotacional do significado dos modelos, e META-LEVEL, que redefine a parte operacional da semântica dos modelos. Os componentes da visão denotacional são apresentados através do módulo IOPT_NET_CONFIGURATION, com visão semântica de elementos pertencentes à linguagem, tais como eventos, sinais, lugares e outros mais. Concentrando-se na visão operacional, no módulo META-RULE-EXTENSION, podemos encontrar o uso de operadores que representam termos e módulos, fornecendo operações eficientes de descendência reduzindo as computações a partir do nível meta para o nível de objetos. Desta forma, definimos casos especiais de computação de regras que não estão na semântica original de reescrita do Maude. Reusando os últimos dois módulos apresentados, temos o módulo META_PETRI_NET, que define as regras de computação para redes IOPT baseando-se na forma de definição de transições das redes de Petri. A computação dos tokens de lugar para lugar através do disparo das transições será gerada para esse módulo. A computação de regras entre os módulos envolvidos será gerada nos módulos COMPONENT_1 até COMPONENT_N. Desta forma, esses módulos definirão a produção e o consumo dos componentes ao se comunicarem caracterizando o paradigma de sincronização que será gerado para estes módulos. Finalmente, complementando a definição da semântica das IOPTs, o módulo

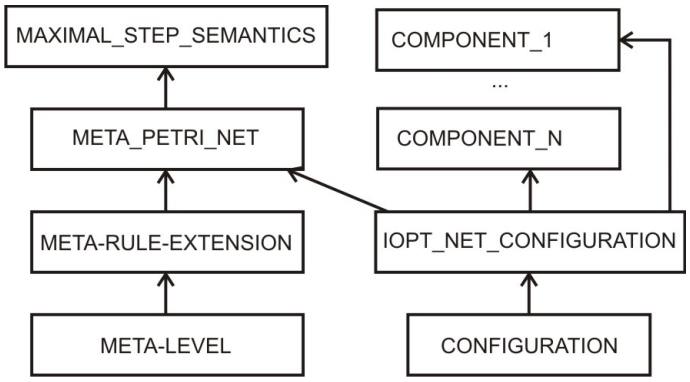

Figura 43: A infraestrutura de módulos para semântica das redes IOPT

MAXIMAL_STEP_SEMANTICS caracteriza a semântica do passo máximo das IOPTs. Assim, operações como *metaApply* do Maude, que casa um termo com o lado esquerdo de uma regra, aplica a regra a partir do topo da estrutura sintática do termo e retorna a meta-representação do termo, serão bastante usadas. Por exemplo, ela pode ser usada para tomar a aplicação de todas ou de qualquer combinação de transições habilitadas concorrentes em um único passo.

Após a execução da Splitting, tomando M_{in} e M_{out}, produzimos a tabela de equivalência apresentada de forma simplificada na Tabela 10. Essa tabela é reusada posteriormente para garantir a equivalência entre as transições durante a verificação de equivalência.

M_{in}	M_{out}
ARRIVE	ARRIVE_M;ARRIVE_S
LEAVE	LEAVE_M;LEAVE_S
OPEN	OPEN_M;OPEN_S

Tabela 10: Tabela de Equivalência para o controlador domótico

Analisemos o modelo da Figura 41 como M_{in} e o da Figura 42 como M_{out}, envolvidos na transformação *Splitting*, que serão utilizados como ilustração para extração dos *modelos semânticos* e determinação de sua *semântica dinâmica*. Por decisões do *Cliente MDA*, o *modelo semântico* de M_{in} teve que ser representado na variação *IOPT com Semântica Maximal Step e Eventos*, enquanto que o *modelo semântico* de M_{out} teve que ser representado na variação *Globalmente Assíncrono e Localmente Síncrono* com características de distribuição.

Os fragmentos de código a seguir ilustram o *modelo semântico* extraído para M_{in}. Ele se inicia no Código 5.7 com o módulo IOPT_NET-CONFIGURATION. Este módulo define os tipos e elementos estruturais básicos para uma rede, tais como marcações de lugares (linha 2) e marcação

de rede e suas propriedades (linhas 3 e 4).

O módulo apresentado no Código 5.8 descreve uma extensão auxiliar ao META-LEVEL do Maude para permitir que trabalhemos com listas ordenadas de regras, e assim, poder controlar melhor o disparo das transições. Temos o operador @ representando a concatenação de listas e isto será bastante útil para esta semântica de execução escolhida, já que o nível META-LEVEL original permite apenas que se trabalhe com conjuntos (sem ordenação) de regras.

Código 5.7 em Maude. Definição da estrutura Rede de Petri do PSM IOPT para M_{in}

```
1: mod IOPT_NET-CONFIGURATION is protecting CONFIGURATION .
2:   sorts Marking Place .
2:   ops waitingToEnable enabled ... : -> Place .
3:   op  empty : -> Place . op  __ : Place Place -> Place [assoc comm id: empty] .
4:   op (_,_, ... ) : Place Place ... -> Marking . op noState : -> Marking [ctor] .
5: endm
```

Código 5.8 em Maude. Definição abstrata do conjunto de regras

```
1: mod META-RULE-EXTENSION is protecting META-LEVEL .
2:   sort RuleList .  subsort Rule < RuleList .  op noRule : -> RuleList [ctor] .
3:   op _@_ : RuleList RuleList -> RuleList [ctor assoc id: noRule] .
4: endm
```

O Código 5.9 apresenta a extensão para a semântica dinâmica das redes de Petri. Na linha 4 temos um exemplo de uma transição chamada `ARRIVE`, com prioridade 0 (zero), e que casa a presença de uma marca (`waitingForPresence`), o substituindo por outra marca (`presence`). A transição nas linhas 5 e 6 também segue esse padrão: tem nome `LEAVE`, prioridade 2 e substitui marcas `enabled` e `presence` por `waitingForPresence` e `waitingToOpen`. Todas as transições seguem este padrão.

Código 5.9 em Maude. Módulo dinâmico do PSM IOPT para M_{in}

```
1: mod META_PETRI_NET is
2:   protecting META-RULE-EXTENSION . protecting IOPT_NET-CONFIGURATION .
3:   vars MP1 MP2 ... : Place .
4:   rl [0-ARRIVE] : (MP1,MP2, waitingForPresence MP3, ...) => (MP1,MP2,MP3, presence,MP4, ...) .
5:   rl [2-LEAVE] : (MP1,enabled MP2,MP3, presence MP4, ...) =>
6:    (MP1,MP2, waitingForPresence MP3, ... ,waitingToOpen MP11) .
...
```

O módulo apresentado no Código 5.10, chamado de `MAXIMAL_STEP`, redefine a semântica de execução das redes de Petri como IOPTs com passo máximo. As linhas 3-9 apresentam diversas operações auxiliares para a concretização desse objetivo. Por exemplo, nas linhas 10-12 definimos

axiomas para dizerem se uma regra deve ser aplicada no contexto dessa semântica dinâmica. Isto deverá ser verdade se a meta aplicação (`metaApply`) dessa regra produzir um resultado que seja diferente de falha (`failure`) (linhas 10-11). Caso contrário, a regra não será aplicada (linha 12). Finalmente, as linhas 13-16 apresentam a regra operacional que redefine essa semântica. Intitulada por `maximal-step`, a partir de uma marcação `I`, e uma String `S` que representa as regras de alcançabilidade, ela reescreve a partir da operação `maxStep` para uma nova marcação de acordo com as novas regras que foram aplicadas pela semântica de passo máximo.

Código 5.10 em Maude. Módulo de extensão da semântica dinâmica para M_{in}

```
1: mod MAXIMAL_STEP_SEMANTICS is protecting META_PETRI_NET .
2:  var I : Marking . var S : String .
...
3:   op applicableRule? : Term Qid Module      -> Bool .
4:   op applicableRules : Term RuleSet Module  -> RuleSet .
5:   op orderRules      : RuleSet String       -> RuleList .
6:   op orderedRules    : Term Module          -> RuleList .
7:   op getRule         : Term Module String   -> RuleList .
8:   op getRulesId      : Module Term RuleList -> String .
9:   op maxStep         : Module Term RuleList -> Marking .
...
10:   ceq applicableRule?(T, Q, M) = true
11:    if R := metaApply(M, T, Q, none, 0) /\ R =/= failure .
12:  eq applicableRule?(T, Q, M) = false [owise] .
...
13:  rl [maximal-step] : (I, S) =>
14:   maxStep(module, upTerm(I), applicableRules(upTerm(I), rules, module)),
15:   getRulesId(applicableRules(upTerm(I), rules, module))
```

O Código 5.11 representa o *modelo semântico* extraído para o PSM representado por M_{out}. A estrutura de estado nesse caso é bem mais complexa, quando comparando ao PSM representado por M_{out}. Temos a definição dos tipos envolvidos, onde aparecem eventos, sinais, seus conjuntos e outros mais (linhas 2-3), definição dos eventos existentes na rede (linhas 4-5), definição dos sinais existentes na rede (linhas 6-7), definição dos lugares (linhas 8-9), composição de conjuntos de sinais, de eventos e marcações como um atributo de um objeto rede (linha 10), composição dos dois objetos que representam as redes particionadas mais a configuração das mensagens como uma estrutura IOPT (linha 11), construtores de conjuntos de eventos e de sinais (linha 12), construtores das mensagens enviadas pelas transições mestres (linha 13) e, finalmente, construtores dos dois módulos de redes particionadas (linha 14).

Código 5.11 em Maude. Definição da estrutura Rede de Petri do PSM IOPT GALS para M_{out}

```
1: mod IOPT_NET-CONFIGURATION is including CONFIGURATION .
2:  sorts Event EventSet Signal SignalSet Marking Place IOPT Message .
3:  subsort Place < Marking .
4:  ops ARRIVE LEAVE OPEN : -> Event . op idle : -> Event .
5:  op __ : Event Event -> Event [assoc comm id: idle] .
6:  ops SW1 SW2 SW3 : -> Signal . op noSignal : -> Signal .
7:  op __ : Signal Signal -> Signal [assoc comm id: noSignal] .
8:  ops waitingToEnable enabled ... : -> Place . op  empty : -> Place .
9:  op  __ : Place Place -> Place [assoc comm id: empty] .
10: op _+_+_ : EventSet SignalSet Marking -> Attribute [ctor gather (& & &)] .
11: op _,_,_ : Object Object Configuration -> IOPT . op none  : -> IOPT .
12: op {_} : Event -> EventSet [ctor] . op [_] : Signal -> SignalSet [ctor] .
13: ops M-ARRIVE M-LEAVE M-OPEN : -> Configuration [ctor] .
14: ops Petri1 Petri2 : -> Cid [ctor] .
...
```

Cada um dos componentes de rede particionados terão seu comportamento representado por um módulo que reusa a estrutura existente em `IOPT_NET-CONFIGURATION`. Por exemplo, o Código 5.12 apresenta a definição de duas transições mestras existentes (`MASTER-ARRIVE` e `MASTER-OPEN`) para o primeiro módulo do PSM de saída provido pela *Splitting*, ambas transições tendo prioridade zero.

Código 5.12 em Maude. Módulo dinâmico do PSM IOPT GALS para Componente 1 de M_{out}

```
1: mod COMPONENT_1 is protecting IOPT_NET-CONFIGURATION .
...
2:    rl [0-MASTER-ARRIVE] : < petri : Petri1 | E + S + waitingForPresence AnyNet >,
3:     < petri2 : Petri2 | {E2} + [S2] + AnyNet2 >, C =>
4:     < petri : Petri1 | E + S + presence AnyNet>,
5:     < petri2 : Petri2 | {ARRIVE E2} + [SW1 S2] + AnyNet2 >, M-ARRIVE C .
6:    rl [0-MASTER-OPEN] : < petri : Petri1 | E + S + waitingToOpen AnyNet >,
7:     < petri2 : Petri2 | {E2} + [S2] + AnyNet2 >, C =>
8:     < petri : Petri1 | E + S + AnyNet >,
9:     < petri2 : Petri2 | {OPEN E2} + [SW3 S2] + AnyNet2 >, M-OPEN C .
```

Código 5.13 em Maude. Módulo dinâmico do PSM IOPT GALS para Componente 2 de M_{out}

```
1: mod COMPONENT_2 is protecting IOPT_NET-CONFIGURATION .
...
2:    rl [2-MASTER-LEAVE] : < petri : Petri1 | {E1} + [S1] +  AnyNet >,
3:     < petri2 : Petri2 | {E2} + [S2] + enabled presence-copy AnyNet2 >, C =>
4:     < petri : Petri1 | {LEAVE E1} + [SW2 S1] +  AnyNet >,
5:     < petri2 : Petri2 | {E2} + [S2] + AnyNet2 >, M-LEAVE C .
```

De maneira complementar, o Código 5.13 apresenta um exemplo de uma transição mestra para o segundo componente gerado para o PSM de saída. A transição chama-se `MASTER-LEAVE`) e possui prioridade 2.

Neste caso, temos que o módulo `META_PETRI_NET`, com fragmento apresentado no Código 32, além de definir as transições que não são de comunicação das redes, conforme apresentado no Código 5.14, agora também incorpora as regras que representam transições de comunicação definidas nos módulos dos componentes particionados (linhas 4-10).

Código 5.14 em Maude. Módulo dinâmico do PSM IOPT para M_{out}

```
1: mod META_PETRI_NET is
2:  protecting META-RULE-EXTENSION . protecting IOPT_NET-CONFIGURATION .
3:  sort PetriNet .
4:  op rulesOne  : -> RuleSet . op moduleOne : -> Module .
5:  op rulesTwo  : -> RuleSet . op moduleTwo : -> Module  .
6:  op (_,_)  : IOPT String -> PetriNet .
7:  eq moduleOne = upModule('COMPONENT_1, false) .
8:  eq rulesOne  = getRls(moduleOne) .
9:  eq moduleTwo = upModule('COMPONENT_2, false) .
10: eq rulesTwo  = getRls(componentTwo) .
...
```

O módulo `MAXIMAL_STEP`, presente no Código 5.15, redefine a semântica de execução das redes para a semântica adequada de passo máximo disponível para sistemas GALS. Dessa forma, cada módulo terá sua própria regra operacional, conforme pode-se perceber nas linhas 2-4, através da regra `maximal-step-one` e linhas 5-7, através da regra `maximal-step-two`.

Código 5.15 em Maude. Módulo de extensão da semântica dinâmica para M_{out}

```
1: mod MAXIMAL_STEP_SEMANTICS is protecting META_PETRI_NET .
...
2:  rl [maximal-step-one] : (I, S) =>
3:   maxStep(moduleOne, upTerm(I), orderedRules(upTerm(I), rulesOne, moduleOne)),
4:   getRulesId(moduleOne, upTerm(I), orderedRules(upTerm(I), rulesOne, moduleOne)) .
5:  rl [maximal-step-two] : (I, S) =>
6:   maxStep(moduleTwo, upTerm(I), orderedRules(upTerm(I), rulesTwo, moduleTwo)),
7:   getRulesId(moduleTwo, upTerm(I), orderedRules(upTerm(I), rulesTwo, moduleTwo)) .
```

A Figura 44 apresenta um diagrama de inclusão dos módulos necessários para construção de uma infraestrutura para verificação dos modelos semânticos. Definimos o módulo TRANSITION_PREDS que define as proposições atômicas dos modelos de acordo com características das redes, como por exemplo, o número de marcas nos lugares. Este módulo inclui também os

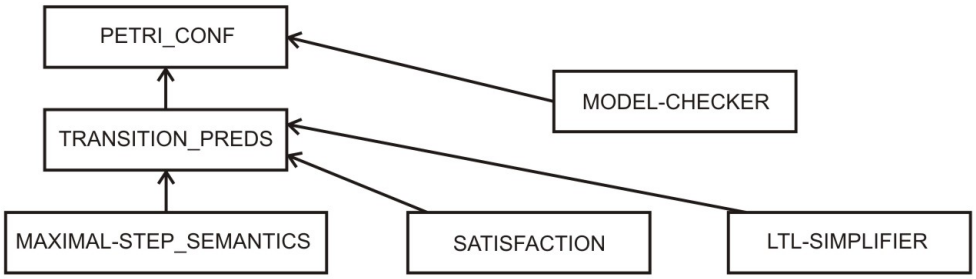

Figura 44: A infraestrutura de módulos para verificação das redes IOPT

módulos pré-definidos do Maude SATISFACTION, para verificação de satisfatibilidade das fórmulas, e LTL-SIMPLIFIER que faz processamento de expressões na lógica temporal LTL. Por fim, o módulo PETRI_CONF define a configuração inicial do modelo sob verificação para que seja processada sua verificação. Além da inclusão TRANSITION_PREDS, ele inclui o módulo MODEL-CHECKER, que é onde os algoritmos de verificação do Maude estão definidos. A seguir, exploraremos esses módulos com um pouco mais de detalhes.

A partir da extração dos *modelos semânticos*, um módulo de propriedades é gerado automaticamente com propriedades *default* de acordo com a habilitação de transições com eventos cadastrados. Dessa forma, por exemplo, para o primeiro modelo temos que o Código 5.16 define todas essas propriedades, conforme estão definidas na linha 4. A linha 6, por exemplo, garante que a propriedade `ARRIVE` existe se existir uma marca `waitingForPresence` na marcação global, a linha 7 garante a propriedade `LEAVE` se existir uma marca `enabled`, uma marca `presence` e uma marca `empty` na marcação global, e, finalmente, a linha 8 define a propriedade `OPEN` pela existência de uma marca `waitingToOpen`.

Código 5.16 em Maude. Módulo default de especificação de propriedades para M_{in}

```
1: mod TRANSITION-PREDS is
2:    protecting MAXIMAL_STEP . protecting SATISFACTION . inc LTL-SIMPLIFIER .
3:    subsort PetriNet < State . subsort Prop < Formula .
4:    ops ARRIVE LEAVE OPEN WARM DISFIREALERT ENABLE ALERT STOP OFF BRIGHT DARK : -> Prop .
5:    vars MP1 MP2 MP3 MP4 MP5 MP6 MP7 MP8 MP9 MP10 MP11 : Place .
6:    eq (MP1, MP2, waitingForPresence MP3, ..., S) |= ARRIVE = true .
7:    eq (MP1, enabled MP2, MP3, presence MP4, MP5, empty, ... , S) |= LEAVE = true .
8:    eq ( ... , MP10, waitingToOpen MP11, S) |= OPEN = true .
...
```

Finalmente, o módulo `PETRI_CONF`, definido no Código 5.17, define a configuração inicial de um modelo a partir do seu estado atual em que foi transformado, e que a partir daí, será o

estado a partir de onde as propriedades temporais serão verificadas. As linhas 3-4 definem a configuração inicial da mesma forma que a rede da Figura 41 está desenhada: com marcas em `waitingToEnable`, `waitingForPresence`, `waitingForFire`, `stopAlert` e `dark`.

Código 5.17 em Maude. Módulo de definição de configuração para M_{in}

```
1: mod PETRI_CONF is inc MODEL-CHECKER . inc TRANSITION-PREDS .
2:  op net : -> PetriNet .
3:  eq net = (waitingToEnable, empty, waitingForPresence, empty, waitingForFire, empty,
4:    stopAlert, empty, dark, empty, empty, "") .
5: endm
```

Código 5.18 em Maude. Verificação de deadlock para M_{in}

```
1: search in PETRI_CONF : net =>! Any:PetriNet .
2: No solution.
3: states: 237  rewrites: 157652 in 530ms cpu (522ms real) (297456 rewrites/second)
```

Código 5.19 em Maude. Verificação de deadlock para M_{out}

```
1: search in PETRI_CONF : net =>! Any:PetriNet .
2: No solution.
3: states: 495  rewrites: 271345 in 1380ms cpu (1395ms real) (196626 rewrites/second)
```

Emprego da Atividade v. *Aplicação do Verificador Formal para expressar a equivalência dos modelos.* Procedemos com a busca por *deadlock* nos dois modelos, como a primeira propriedade a ser verificada, conforme ilustrado nos Códigos 5.18 e 5.19.

A verificação da primeira propriedade, derivada pelo padrão *Existência* entre eventos apresenta a resposta negativa esperada para ambos os modelos. Nos códigos 5.20 e 5.21, pode-se perceber que essas especificações em linguagem de alto nível foram automaticamente traduzidas para fórmulas LTL bem mais complexas. Dessa forma, temos a geração do padrão [](!Q) | <>(Q & <>P)), conforme especifica esse padrão. Os contra-exemplos produzidos foram omitidos dos códigos apresentados, mas estão ilustrados na Figura 45 para simplificar o entendimento. A generalização empregada para garantia da verificação de equivalência é muito simples, parte do princípio da detecção do menor final demonstrador do contra-exemplo. Observando ambos contra-exemplos em 45(a) e 45(b), percebemos que um círculo de eventos DARK-BRIGHT-DARK se formou para os dois casos, garantindo a equivalência mesmo no caso de não-satisfação das propriedades. Novamente, enfatizamos que essa estrutura de prova é comum em todas as aplicações de transformações abordadas por este trabalho.

Figura 45: Contra-exemplos produzidos pela verificação de equivalência

Código 5.20 em Maude. Verificação do Padrão Existência/Após para M_{in}

```
1: reduce in PETRI_CONF : modelCheck(net, []~ ENABLE \/ <> (ENABLE /\ <> ALERT)) .
2: rewrites: 580 in 10ms cpu (4ms real) (58000 rewrites/second)
3: result ModelCheckResult: counterexample(...)
```

Código 5.21 em Maude. Verificação do Padrão Existência/Após para M_{out}

```
1: reduce in PETRI–CONF : modelCheck(net, []~ ENABLE \/ <> (ENABLE /\ <> ALERT)) .
2: rewrites: 282 in 10ms cpu (8ms real) (28200 rewrites/second)
3: result ModelCheckResult: counterexample(...)
```

Por fim, ilustramos detalhadamente a verificação sob um padrão de especificação por completo. O padrão escolhido foi o de Precedência, conforme especifica esse padrão. Esse padrão busca especificar informações sobre variações dado o cenário de que uma determinada propriedade S precede um propriedade P. Interessou ao *cliente MDA* considerar S = ENABLE e P = ALERT, o que contrariamente ao exemplo anterior assume sempre o valor verdade. A partir desse cenário, cinco tipos de consultas são possíveis de acordo com o que especifica esse padrão. Os códigos

5.22 e 5.23 ilustram os resultados para o primeiro caso, onde busca-se saber se globalmente esse padrão é válido. A fórmula derivada é !P W S. Valores positivos são obtidos por essa verificação de equivalência no Maude model-checker.

Código 5.22 em Maude. Verificação do Padrão Precedência/Globalmente para M_{in}

```
1: reduce in PETRI_CONF : modelCheck(net, ~ ALERT W ENABLE) .
2: rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
3: result Bool: true
```

Código 5.23 em Maude. Verificação do Padrão Precedência/Globalmente para M_{out}

```
1: reduce in PETRI–CONF : modelCheck(net, ~ ALERT W ENABLE) .
2: rewrites: 28 in 0ms cpu (0ms real) (~ rewrites/second)
3: result Bool: true
```

O segundo caso aplica a consulta *Antes de R* a esse padrão. Tomando R = STOP, queremos saber se ENABLE precede ALERT antes de STOP. Essa propriedade também é de interesse do cliente final da solução e é derivada como <>R -> (!P U (S | R)). Seus resultados da verificação de equivalência são apresentados nos Códigos 5.24 e 5.25.

Código 5.24 em Maude. Verificação do Padrão Precedência/Antes para M_{in}

```
1: reduce in PETRI_CONF : modelCheck(net, <> STOP -> ~ ALERT U ENABLE \/ STOP) .
2: rewrites: 45 in 0ms cpu (0ms real) (~ rewrites/second)
3: result Bool: true
```

Código 5.25 em Maude. Verificação do Padrão Precedência/Antes para M_{out}

```
1: reduce in PETRI–CONF : modelCheck(net, <> STOP -> ~ ALERT U ENABLE \/ STOP) .
2: rewrites: 45 in 0ms cpu (0ms real) (~ rewrites/second)
3: result Bool: true
```

O terceiro caso aplica a consulta *Após Q* a esse padrão. Tomando Q = OPEN, queremos saber se ENABLE precede ALERT após OPEN. Essa propriedade é derivada como []!Q | <>(Q & (!P W S)). Seus resultados da verificação de equivalência são apresentados nos Códigos 5.26 e 5.27.

Código 5.26 em Maude. Verificação do Padrão Precedência/Depois para M_{in}

```
1: reduce in PETRI_CONF : modelCheck(net, <> (OPEN /\ (~ ALERT W ENABLE)) \/ []~ OPEN) .
2: rewrites: 121547 in 380ms cpu (539ms real) (319860 rewrites/second)
3: result Bool: true
```

Código 5.27 em Maude. Verificação do Padrão Precedência/Depois para M_{out}

```
1: reduce in PETRI-CONF: modelCheck(net, <>(SLAVE-OPEN /\ (~ ALERT W ENABLE)) \/ []~ SLAVE-OPEN) .
2: rewrites: 66122 in 1920ms cpu (3729ms real) (34438 rewrites/second)
3: result Bool: true
```

O quarto caso aplica a consulta *Entre Q e R* a esse padrão. Tomando Q = OPEN e R = STOP, queremos saber se ENABLE precede ALERT entre OPEN e STOP. Essa propriedade é derivada como [](Q & !R & <>R) -> (!P U (S | R))). Seus resultados da verificação de equivalência são apresentados nos Códigos 5.28 e 5.39.

Código 5.28 em Maude. Verificação do Padrão Precedência/Entre para M_{in}

```
1: reduce in PETRI_CONF : modelCheck
2:  (net, [](<> STOP /\ (OPEN /\ ~STOP) -> ~ ALERT U ENABLE \/ STOP)) .
3: rewrites: 125679 in 400ms cpu (399ms real) (314197 rewrites/second)
4: result Bool: true
```

Código 5.29 em Maude. Verificação do Padrão Precedência/Entre para M_{out}

```
1: reduce in PETRI-CONF : modelCheck
2:  (net, [](<> STOP /\ (SLAVE-OPEN /\ ~ STOP) -> ~ ALERT U ENABLE \/ STOP)) .
3: rewrites: 70418 in 1910ms cpu (4017ms real) (36868 rewrites/second)
4: result Bool: true
```

Finalmente, o quinto caso aplica a consulta *Após Q até R* a esse padrão. Tomando Q = OPEN e R = STOP, queremos saber se ENABLE precede ALERT após OPEN até que STOP aconteça. Essa propriedade é derivada como [](Q & !R -> (!P W (S | R))). Seus resultados da verificação de equivalência são apresentados nos Códigos 5.30 e 5.31.

Código 5.30 em Maude. Verificação do Padrão Precedência/Após e Até para M_{in}

```
1: reduce in PETRI_CONF : modelCheck
2:  (net, [](OPEN /\ ~ STOP -> ~ ALERT W ENABLE \/ STOP)) .
3: rewrites: 125686 in 390ms cpu (391ms real) (322271 rewrites/second)
4: result Bool: true
```

Código 5.31 em Maude. Verificação do Padrão Precedência/Após e Até para M_{out}

```
1: reduce in PETRI-CONF : modelCheck(net, [](SLAVE-OPEN /\ ~ STOP -> ~ ALERT W ENABLE \/ STOP)) .
2: rewrites: 70425 in 1940ms cpu (4160ms real) (36301 rewrites/second)
3: result Bool: true
```

5.3 Aplicação de uma Transformação PIM-para-PSM

Nesta seção, apresentamos um contexto em que empregamos a verificação de equivalência entre modelos em uma transformação PIM-para-PSM. Após uma apresentação da motivação e do cenário do trabalho, nos concentramos nas características em que a transformação *Splitting*, que é a transformação sob questão, adicionou aos modelos de saída e vamos direto ao *emprego da atividade v* da verificação, pois os passos anteriores de extração dos *modelos semânticos* de PIMs e PSMs já foram amplamente discutidos nas seções anteriores.

Devido ao fato de a maioria dos sistemas embarcados adquirirem características de portabilidade, passarem a ter um projeto voltado para execução de funções dedicadas e terem requisitos de computação de tempo real, a captura dos requisitos no alto nível e a representação de conceitos e comportamento no baixo nível precisam estar sempre atados, de uma maneira preservadora de semântica. Além do mais, avanços no paradigma de interação dessas tecnologias visam a integração desses sistemas e a comunicação de diferentes aplicações. Neste cenário, *web-services*, que podem ser especificados em WS-BPEL (Web-Services Business Process Execution Language) [Andrews et al., 2003], permitindo a comunicação através do formato XML com a Arquitetura Orientada a Serviço (SOA) [Josuttis, 2007], é uma opção bem consolidada para ganhos de agilidade e de eficiência ao colocar recursos diferentes para interagir, casando, de forma equilibrada, requisitos de alto nível com representações de baixo nível.

Diversos trabalhos tem empregado técnicas de mapeamento de WS-BPEL para redes de Petri no intuito de reusar todas as vantagens dessa linguagem, que já foram mencionadas nesse trabalho. Neste caso, as oWFnets (Open workflow nets) [Stahl, 2005], um tipo especial de redes de Petri, que generalizam *workflows* ao introduzirem uma interface para passagem de mensagens assíncronas, que permitem modelar serviços e suas interações através de canais de lugares. Por outro lado, ao baixo nível de especificação de sistemas, temos que as IOPTs são mapeadas para hardware ou software, de acordo com métricas específicas de custo e desempenho, usando técnicas de codesign. Desta forma, a transformação *Splitting* pode combinar esses dois lados [Barbosa et al., 2010a], resolvendo um problema comum da engenharia de sistemas: a maioria das metodologias não resolveram a falta de comunicação existente entre artefatos independentes e específicos de plataforma de forma apropriada [Stahl et al., 2006]. Nesta seção, analisamos preservação de semântica vendo a *Splitting* como uma transformação PIM-para-PSM.

5.3.1 Definição dos Modelos

Escolhemos um cenário de avaliação com uma complexidade considerada simples. Tomamos um modelo de serviço de leilão, como proposto no documento de especificação da linguagem WS-BPEL [Alves, 2006]. Ele é baseado em múltiplas atividades que criam instancias de um processo. Esse processo deve coletar informação de um vendedor e um comprador com tempos de chegada aleatórios, apresentando propriedades interessantes tratadas por essa solução. Através da transformação proposta pelo *Informatics Group de Berlin* [Lohmann, 2007], obtemos um modelo gerado em oWFNet como independente de plataforma, a partir de agora chamado apenas de uma rede de Petri. A transformação *Splitting* adiciona diversas informações de baixo nível, tais como eventos de sistema e sinais, semânticas de execução específicas, modelos de comunicação, além dos modelos estruturalmente particionados.

A Figura 46 apresenta a rede de Petri gerada a partir de um modelo WS-BPEL de entrada utilizando a ferramenta BPEL2oWFnet [Lohmann, 2007]. Essa figura mostra que a rede contém apenas componentes estruturais, com as mensagens de comunicação e outros aspectos mais independentes de plataforma.

Para que as técnicas do FORDESIGN possam ser aplicadas a esse modelo, é necessário tomar atenção especial à nova semântica de execução desejada para o modelo de saída, justificando o emprego da técnica MDA-Veritas. A *Splitting* é capaz de criar componentes que podem ser implementados separadamente e comunicarem-se através de canais de comunicação síncronos direcionados. A estrutura do modelo rede de Petri é transformada em um componente adicionando-se as características de comunicação de uma maneira preservadora de semântica. Na representação PIM, não há representação desse canais de comunicação. As interfaces são representadas por lugares onde as marcas devem aparecer e desaparecer por alguma semântica específica e que não foi representada no modelo. Por este motivo, isso foi descartado e assume-se essas redes como lugar/transição com uma semântica conforme a tratada na Seção 5.1.

Através da Splitting, tem-se um desacoplamento por uma representação síncrona de comunicação entre os componentes do sistema. Isto se deve à introdução de canais de comunicação síncronos diretos através de fusão de transições quando for necessário. Desta forma, interfaces serão transições que geram ou recebem eventos e obtém-se um modelo pronto para uma geração de código automática. Temos os conceitos de sinais e eventos necessários para IOPTs, enriquecendo o modelo conforme apresentado na Figura 47.

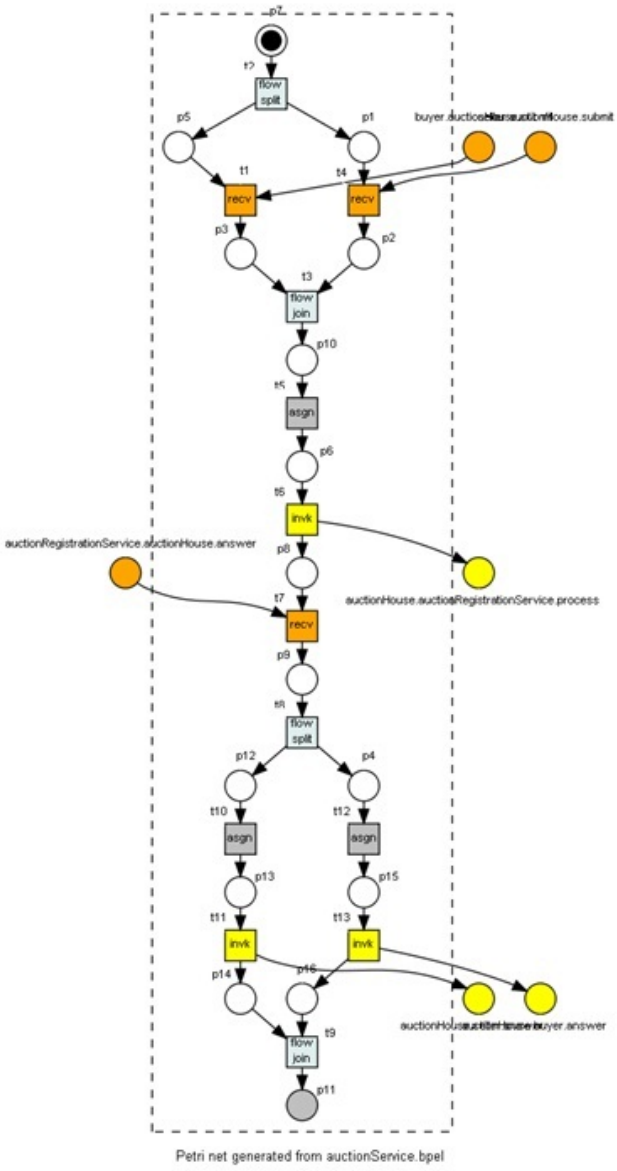

Figura 46: Representação rede de Petri do modelo de serviço de leilão

Características Adicionadas aos Modelos pela Transformação Splitting

Diversos conceitos semânticos específicos de plataforma aparecem nos modelos após a transformação Splitting. Por exemplo, a idéia de representar a interface do componente através de transições permite *macro-transições*, ou seja, a interface se conecta a outra interface como transição através de um lugar, ou uma troca de mensagem, representando o meio, e conectando ambos componentes ao mesmo nível. Assim, os lugares que estavam dentro do componente ficaram preservados e encapsulados, pertencendo à estrutura interna desse componente. A semântica baseada na passagem de marcas entre os lugares é preservada sem nenhuma modificação como aconteceria

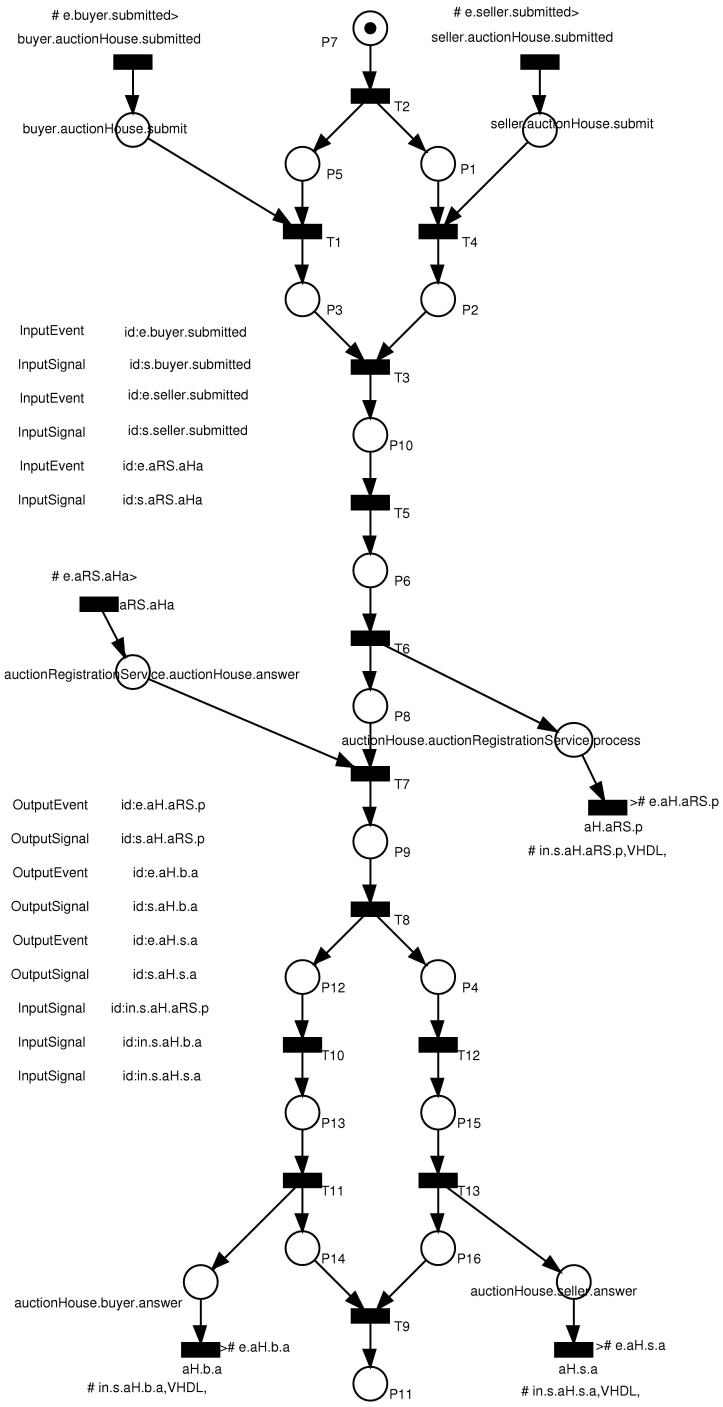

Figura 47: Representação do modelo de serviço de leilão em IOPT

com a fusão de lugares. A Figura 48 ilustra bem esse tipo de regra. Observe que ao centro da figura reusamos o mesmo modelo apresentado na Figura 47 com os novos componentes inseridos nas laterais. Isto é o primeiro critério a ser verificado para equivalência entre redes de Petri como oWFnets e IOPTs. Temos que a comunicação assíncrona e a fusão de lugares foram substituídos

pelos mecanismos de comunicação entre componentes das IOPTs.

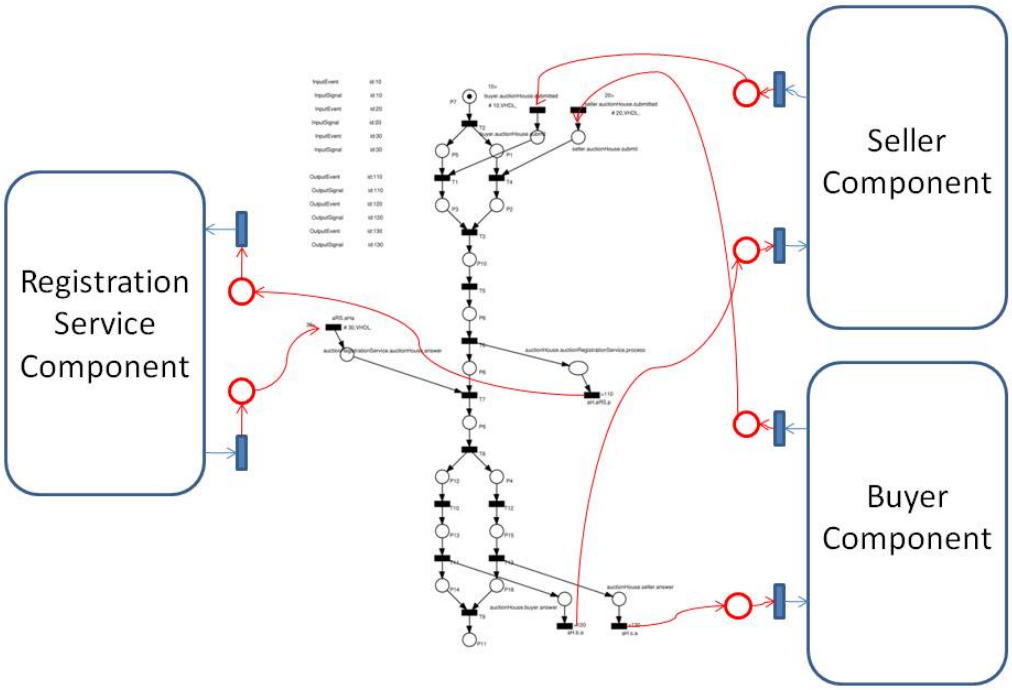

Figura 48: Conexão entre componentes de acordo com uma semântica de IOPT

Outros conceitos semânticos específicos de plataforma também são adicionados aos modelos pela transformação. Entre eles, enumeramos:

- *Descrição das mensagens de sistema*. Em um modelo oWFnet, mensagens não estão explícitas no modelo. Isto foi provido através da aplicação da transformação *Splitting* também. O particionamento do modelo torna explícito que transições estarão esperando por uma ocorrência de um evento. Verificaremos que isto também preservará a semântica de execução da rede sem nenhuma alteração. Não faz-se necessária uma nova definição semântica para os componentes de entrada e de saída como ocorre nas redes oWFnets. Por exemplo, na Figura 49, eventos e sinais associados às transições de comunicação são apresentados ilustrando para duas transições. Como sabemos, estes sinais e eventos são associados diretamente às transições. No exemplo apresentado na Figura 49, um sinal de entrada é definido *in.s.aH.b.a* associado a uma transição de saída *aH.b.a* na qual é lida por outro componente que tem um evento de entrada associado. A transição de entrada *buyer.auctionHouse.submitted* tem um

evento de entrada associado *e.buyer.submitted* que acopla-se através do canal de comunicação a um evento de saída gerado por outro componente.

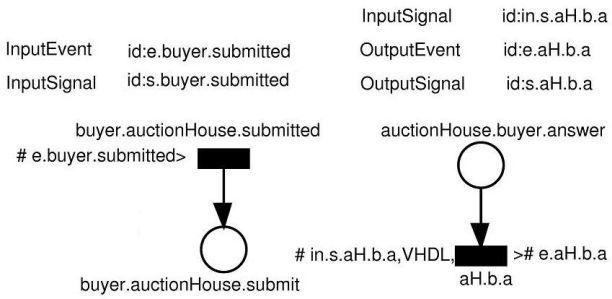

Figura 49: Descrição das mensagens de sistema existentes em IOPT

- *Eventos para processar mudanças de estados*. O recebimento de eventos externos IOPT podem modelar ao nível específico de plataforma estados providos pelo PIM. Por exemplo, o estado *Wait* é modelado com um lugar no qual uma transição associada espera por um evento externo de entrada/saída, e o estado *Fault* é capturado por mensagem enviada aos componentes específico decompostos que estejam processando essa falta.

- *Descrição de um monitor embutido para plataforma síncrona*. O fluxo interno de processo extraído do PIM pode necessitar de mais particionamento para refinamentos devido às características paralelas explícitas. Isto é bastante comum hoje em dia devido ao desaparecimento de arquiteturas monoprocessadas. Neste caso, a comunicação síncrona torna ótimo o uso dos processadores envolvidos refletindo em melhores soluções de projeto.

5.3.2 Definição das Propriedades a Serem Observadas nos Modelos

As propriedades desse cenário classificam-se em duas categorias: (i) preservação de uma propriedade básica; e (ii) preservação de uma propriedade complexa derivada através dos padrões de especificação de propriedades. A propriedade básica mencionada em (i) é o deadlock. A semântica de execução dos componentes em ambos os modelos fazem com que o fluxo em algum momento pare. Para (ii), uma propriedade bastante complexa que interessa preservação ao *cliente MDA* seria a *Precedência Encadeada* (ver Tabela no Capítulo 2.4.5), observando-se os eventos de chegada do comprador e do vendedor do leilão. De fato, o leilão só deveria ser processado após a chegada e oferecimento de propostas desses dois atores.

Em termos técnicos, de acordo com o Código 5.32, deve-se esperar pela chegada de mensagens nas transições `buyer.auctionHouse.submitted` (ou simplificado para `buyer`) e `seller.auctionHouse.submitted` (ou simplificado para `seller`) precedam obrigatoriamente o início do leilão, que é representado pelo lugar `P9`.

Código 5.32 em Maude. Rede de Petri que representa o modelo de serviço de leilão M_{in}

```
1: mod PETRI_NET is sorts Place Marking . subsort Place < Marking .
2:    ops P1 P2 ... P15 P16 Buyer Seller AR AH AHBuyer AHSeller :-> Place .
3:    op empty : -> Marking . op __ : Marking Marking -> Marking [assoc comm id: empty] .
4:    op _,_,_,_,_,_ : Marking Marking Marking Marking Marking Marking -> Marking .
5:    vars AM1 AM2 AM3 AM4 AM5 AM6 : Marking .
6:    rl [T1] : ((Buyer AM1), AM2, AM3, AM4, AM5, AM6) P5 => (AM1, AM2, AM3, AM4, AM5, AM6) P3 .
7:    rl [T2] : P7 => P1 P5 .
8:    rl [T3] : P3 P2 => P10 .
9:    rl [T4] : (AM1, Seller AM2, AM3, AM4, AM5, AM6) P1 => (AM1, AM2, AM3, AM4, AM5, AM6) P2 .
10:   rl [T5] : P10 => P6 .
11:   rl [T6] : (AM1, AM2, AM3, AM4, AM5, AM6) P6 => P8 (AM1, AM2, AM3, AH AM4, AM5, AM6) .
12:   rl [T7] : (AM1, AM2, AR AM3, AM4, AM5, AM6) P8 => (AM1, AM2, AM3, AM4, AM5, AM6) P9 .
13:   rl [T8] : P9 => P12 P4 .
14:   rl [T9]  : P14 P16 => P11 .
15:   rl [T10] : P12 => P13 .
16:   rl [T11] : (AM1,AM2,AM3,AM4,AM5,AM6) P13 => (AM1,AM2,AM3,AM4,AHBuyer AM5,AM6)  P14 .
17:   rl [T12] : P4 => P15 .
18:   rl [T13] : (AM1,AM2,AM3,AM4,AM5,AM6) P15 => (AM1,AM2,AM3,AM4,AM5,(AHSeller AM6)) P16 .
19:   rl [buyer] : empty,AM2,AM3,AM4,AM5,AM6 => Buyer,AM2,AM3,AM4,AM5,AM6 .
20:   rl [seller] : AM1,empty,AM3,AM4,AM5,AM6 => AM1,Seller,AM3,AM4,AM5,AM6 .
21:   rl [ar] : AM1,AM2,empty,AM4,AM5,AM6 => AM1,AM2,AR,AM4,AM5,AM6 .
22:   rl [ah] : AM1,AM2,AM3,AH AM4,AM5,AM6 => AM1,AM2,AM3,AM4,AM5,AM6 .
23:   rl [ah-buyer] : AM1,AM2,AM3,AM4,AHBuyer AM5,AM6 => AM1,AM2,AM3,AM4,AM5,AM6 .
24:   rl [ah-seller] : AM1,AM2,AM3,AM4,AM5,AHSeller AM6 => AM1,AM2,AM3,AM4,AM5,AM6 .
25:   rl [self] : AM1 => AM1 .
endm
```

5.3.3 Emprego da MDA-Veritas

A apresentação do procedimento de verificação de equivalência para este caso PIM-para-PSM foi bastante simplificada, pelo fato de já termos *equações semânticas* bem definidas para extração dos *modelos semânticos* para as redes de Petri da Figura 46, como M_{in}, e da Figura 47, como M_{out}. Aplicamos, de forma direta, a extração dos *modelos semânticos* que estão apresentados com algumas simplificações nos Códigos 5.33 e 5.34. Eles foram gerados para redes de Petri e IOPTs na plataforma GALS respectivamente.

O Código 5.33 recebe simplificações devido à notação para representação de transições na plataforma GALS requisitar uma notação sintática bem mais complexa do que o caso anterior. Neste caso, apenas algumas transições estão ilustradas, sendo as demais facilmente dedutíveis.

Código 5.33 em Maude. Rede de Petri que representa um componente do modelo de serviço de leilão M_{out}

```
1: mod COMPONENT_1 is protecting IOPT_NET-CONFIGURATION .
2:    var petri : Oid . var E : EventSet . vars E1 E2 : Event .
3:    var S : SignalSet . vars S1 S2 : Signal .
4:    var AnyNet : Marking . var O : Object . var C : Configuration .
...
5:    rl [T1] :
6:     <petri:Petri | E+S+ ((Buyer AM1), AM2,AM3,AM4,AM5,AM6) P5 AnyNet>, C =>
7:     <petri:Petri | E+S+ (AM1, AM2, AM3, AM4, AM5, AM6) P3 AnyNet>, C .
8:    rl [T2]  :
9:     <petri:Petri | E+S+ P7 AnyNet>, C =>
10:    <petri:Petri | E+S+ P1 P5 AnyNet>, C .
...
11:   rl [buyer] :
12:    <petri:Petri | E + S + (empty,AM2,AM3,AM4,AM5,AM6) AnyNet>, C =>
13:    <petri:Petri | E + S + (Buyer,AM2,AM3,AM4,AM5,AM6) AnyNet >, C .
14:   rl [seller] :
15:    <petri:Petri | E + S + (AM1,empty,AM3,AM4,AM5,AM6) AnyNet >, C =>
16:    <petri:Petri | E + S + (AM1,Seller,AM3,AM4,AM5,AM6) AnyNet >, C .
...
```

Código 5.34 em Maude. Definição de propriedades básicas para M_{in}

```
1: mod TRANSITION-PREDS is
2:    protecting PETRI_NET . protecting SATISFACTION . inc LTL-SIMPLIFIER .
3:    subsort Marking < State . subsort Prop < Formula .
4:    ops BUYER SELLER AR AH AH-BUYER AH-SELLER : -> Prop .
5:    vars AM1 AM2 AM3 AM4 AM5 AM6 : Marking .
6:    eq empty,AM2,AM3,AM4,AM5,AM6 |= BUYER = true .
7:    eq AM1,empty,AM3,AM4,AM5,AM6 |= SELLER = true .
8:    eq AM1,AM2,empty,AM4,AM5,AM6 |= AR = true .
9:    eq AM1,AM2,AM3,AH AM4,AM5,AM6 |= AH = true .
10:   eq AM1,AM2,AM3,AM4,AHBuyer AM5,AM6 |= AH-BUYER = true .
11:   eq AM1,AM2,AM3,AM4,AM5,AHSeller AM6 |= AH-SELLER = true .
12:   eq P9 |= SUBMISSION-DONE = true .
13:   eq P14 P16 |= FINISHED = true .
...
14:endm
```

O Código 5.34 ilustra algumas propriedades primitivas que são interessantes para a verificação definida pelo *cliente MDA*. Apresentamos propriedades que interessam preservação de semântica

no contexto de confiabilidade da transformação, mas também, para desmistificar a técnica, apresentamos um exemplo de uma propriedade bastante intuitiva que não é preservada devido às diferenças entre as semânticas de execução e que sua não-preservação já era completamente esperada pelo *Cliente MDA*.

Os espaços de estados desses sistemas apresentam características bem diferentes. Por exemplo, para o *modelo semântico* de M_{in} temos pelo menos 408 estados, enquanto que para M_{out} temos apenas 11 estados, conforme é ilustrado na Figura 50. Neste sentido, pergunta-se: como é possível preservação de semântica? A resposta, como apresentada em toda a tese, e para concluirmos este capítulo de explanação detalhada da técnica, pode ser resumida como: *de acordo com as propriedades que sejam de interesse do Cliente MDA*.

Código 5.35 em Maude. Verificação de deadlock para M_{in}

```
1: search in PETRI_CONF : net =>! Any:Marking .
2:  Solution 1 (state 407)
3:  states: 408  rewrites: 1373 in 3570332000ms cpu (74ms real) (0 rewrites/second)
4:  Any:Marking --> P11 (Buyer,Seller,AR,empty,empty,empty)
5:  No more solutions.
6:  states: 408  rewrites: 1373 in 3570332000ms cpu (75ms real) (0 rewrites/second)
```

Código 5.36 em Maude. Verificação de deadlock para M_{out}

```
1: search in PETRI_CONF : net =>! Any:Net .
2: Solution 1 (state 10)
3: states: 11  rewrites: 8805 in 3312284000ms cpu (1105ms real) (0 rewrites/second)
4: Any:Net -->
5:  < petri : Petri | {idle} + [noSignal] + P11 (Buyer,Seller,AR,empty,empty,empty)>,
6:  (none).Configuration,"ah-seller ah-buyer T9 "
7: No more solutions.
8: states: 11  rewrites: 8805 in 3312284000ms cpu (1106ms real) (0 rewrites/second)
```

Código 5.37 em Maude. Precedência encadeada para primeiras ações concorrentes de M_{in}

```
1: reduce in PETRI_CONF : modelCheck(net, <> SUBMISSION-DONE ->
2:  ~SUBMISSION-DONE U O (~SUBMISSION-DONE U SELLER) /\ (BUYER /\ ~SUBMISSION-DONE)) .
3: rewrites: 1499 in 10580266205ms cpu (169ms real) (0 rewrites/second)
4: result Bool: true
```

Aplicando a verificação da propriedade de deadlock, obtemos a verificação de equivalência desejada conforme apresentam os Códigos 5.35 e 5.36. Nestes códigos, observamos que, apesar do número de estados, de reescritas e outros fatores mais serem diferentes, há um resultado lógico equivalente e solução de testemunho alcançando estados também equivalentes de prova.

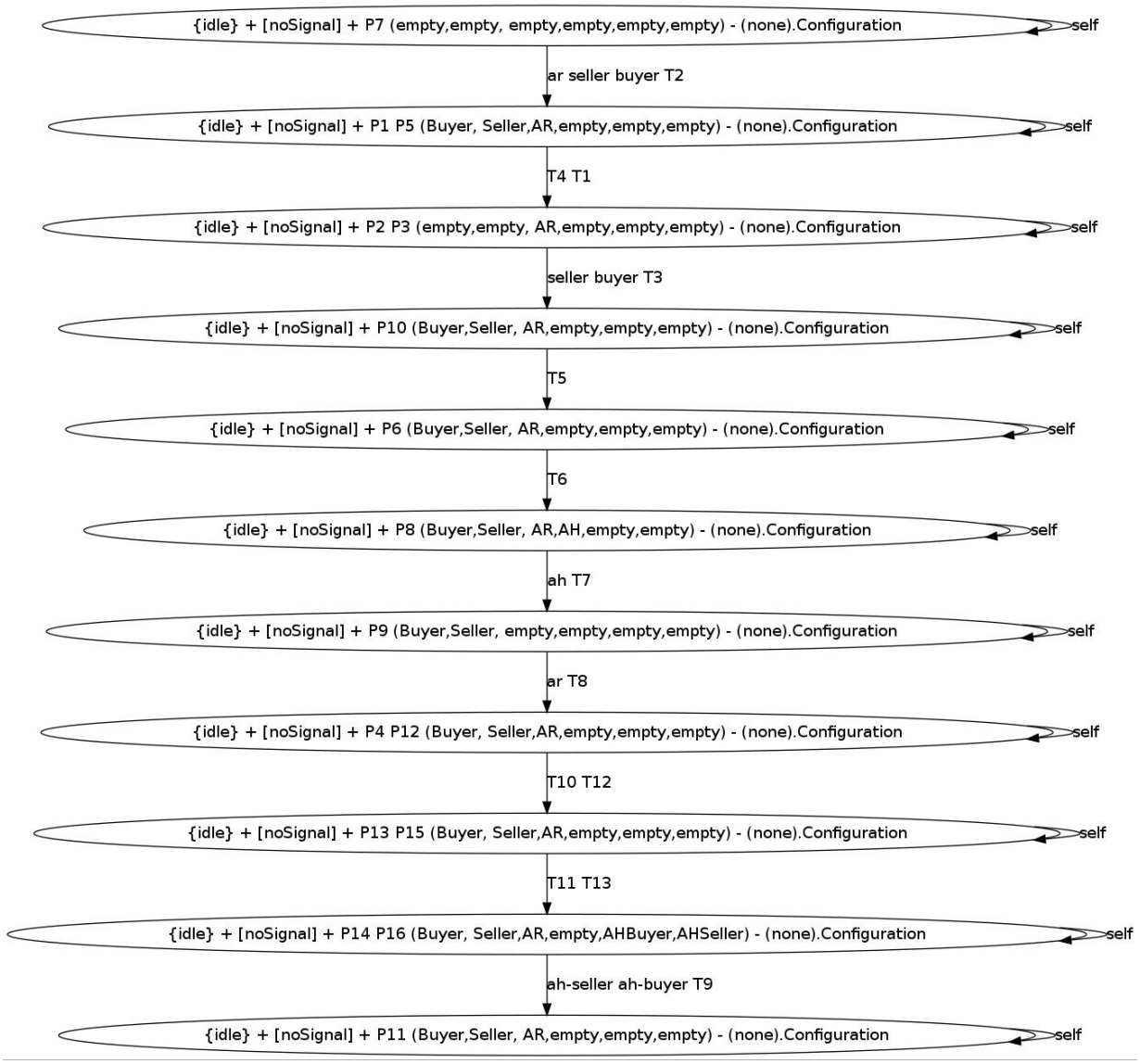

Figura 50: Espaço de estados para o componente do modelo de serviço de leilão M_{out}

Os Códigos 5.37 e 5.38 apresentam para o padrão *Precedência Encadeada/Globalmente* (<>P -> (!P U (S & !P & o(!P U T))), com S = SELLER, T = BUYER e P = SUBMISSION-DONE, os resultados da verificação de equivalência.

Código 5.38 em Maude. Precedência encadeada para primeiras açoes concorrentes de M_{out}

```
1: reduce in PETRI_CONF : modelCheck(net, <> SUBMISSION-DONE ->
2:  ~SUBMISSION-DONE U O (~SUBMISSION-DONE U SELLER) /\ (BUYER /\ ~SUBMISSION-DONE)) .
3: rewrites: 1655 in 15448023825ms cpu (404ms real) (0 rewrites/second)
4: result Bool: true
```

5.4 Recapitulação e Considerações

Este capítulo ilustrou o uso da MDA-Veritas na verificação de uma transformação fundamental do projeto de sistemas embarcados no contexto de modelos PIMs e PSMs, demonstrando a instanciação existente para aquele domínio. A solução detectou os itens envolvidos na preservação de semântica. Além do mais, mencionamos ainda os casos extraordinários de verificação de propriedades que não condizem com a especificação dada pelo *cliente MDA*. Garantimos que ainda assim ocorrerá a detecção de preservação de semântica, mesmo que ocorra de algumas propriedades não serem preservadas, onde devem ser descartadas do conjunto de avaliação.

Para detalhes complementares sobre a formulação e implementação no framework MDA da transformação investigada, o leitor pode consultar [Gomes and Costa, 2007] e [Barbosa et al., 2009b] respectivamente. Para uma continuação da discussão sobre a aplicação em uma transformação PIM-para-PIM, o leitor pode consultar [Barbosa et al., 2009a]. Informações sobre a verificação em plataformas específicas que estiveram presentes na aplicação da transformação PSM-para-PSM podem ser encontrados em [Barbosa et al., 2011] e [Moutinho et al., 2011]. Finalmente, a concepção da idéia que mais tarde resultou na aplicação da transformação PIM-para-PSM está presente em [Barbosa et al., 2010a], apresentando uma contribuição à informática industrial. Outros detalhes sobre esse material, bem como maiores detalhes sobre implementação de artefatos, podem ser encontrados no repositório oficial do projeto, estabelecido em [MDA-VERITAS, 2011].

6 Avaliação Experimental

Para avaliarmos a aplicação da arquitetura MDA-Veritas em cenários de verificação de transformações, optamos por adotar o modelo GQM [Basili et al., 1994], que provê bom apoio à maneira como realizar uma avaliação sobre uma arquitetura, algum produto ou processo. Este capítulo apresenta os resultados desta avaliação em um formato estruturado, visando um melhor entendimento do método sem envolvimento em algumas de suas complexidades.

GQM, que significa o acrônimo de Objetivos (Goals), Questions (Questões) e Métricas (Metrics) é uma abordagem para avaliar soluções em software que define um modelo de medições em três níveis:

1. Conceitual: onde objetivos são definidos para um objeto por uma série de razões, a partir de níveis de qualidade, de pontos de vista e de acordo com o ambiente no qual este objeto esteja inserido;

2. Operacional: onde questões são formuladas para definir modelos do objeto de estudo para que a partir daí possamos focar no objeto para caracterizar o alcance de um objetivo específico;

3. Quantitativo: onde um conjunto de métricas, baseadas nesses modelos, são definidas e associadas com cada questão para que respostas sejam obtidas.

Estruturamos este capítulo da seguinte forma: a Seção 6.2 estabelece toda a formalização e estruturação para o estudo experimental e a Seção 6.3 aplica o modelo definido para avaliação detalhada no contexto de cenários de aplicação da transformação, que são o PIM-para-PIM e o PSM-para-PSM. Esses resultados também podem ser reusados em um cenário de avaliação PIM-para-PSM, já que resultados foram coletados nos dois casos anteriores, ou seja, específico para modelos PIM e para modelos PSM.

6.1 Formalização da Abordagem

Pergunta da Pesquisa

A solução MDA-Veritas, instanciada para o domínio de sistemas concorrentes, foi capaz de verificar equivalência entre modelos PIM e PSM de uma forma eficiente e eficaz para as linguagens de descrição redes de Petri e IOPTs e suas respectivas representações de semântica de execução?

Proposições de Estudo

Tendo por base a aplicação de um conjunto de estudos de caso, caracterizar:

1. A complexidade temporal de aplicação da solução;
2. A complexidade espacial de aplicação da solução;
3. As falhas de equivalência sintática encontradas pela aplicação da solução;
4. As falhas de equivalência semântica encontradas pela aplicação da solução.

Objetivos, Questões e Métricas

Objetivo 1. Medir a característica temporal de desempenho da solução em todo o processo de verificação de equivalência semântica nas transformações MDA.

- Propósito: medição;
- Questão: caracterização temporal;
- Objeto: desempenho da solução em todo o processo de verificação;
- Ponto de vista: provedor da solução.
- Questão 1.1. Qual é o tempo necessário para aplicar cada uma das transformações MDA envolvidas no estudo de caso e sua respectiva construção da tabela sintática?
  - Métrica 1.1.a. TT = TT_{out} - TT_{in};
    Onde TT = Tempo total para aplicação da transformação MDA, TT_{in} = Tempo inicial antes da aplicação da transformação e TT_{out} = Tempo final após a aplicação da transformação.
  - Métrica 1.1.b. TTS = TTS_{out} - TTS_{in};
    Onde TTS = Tempo total para extração da tabela sintática, TTS_{in} = Tempo inicial antes da extração da tabela sintática e TTS_{out} = Tempo final após a extração da tabela sintática.
- Questão 1.2. Qual é o tempo necessário para executar extrações de modelos semânticos?

  - Métrica 1.2.a. TEMS = TEMS_{out} - TEMS_{in};

    Onde TEMS = Tempo total para extração dos modelos semânticos, TEMS_{in} = Tempo inicial antes da extração do modelo semântico e TEMS_{out} = Tempo final após a extração do modelo semântico.

- Questão 1.3. Em que ordem de complexidade temporal está a função do tempo de execução das extrações de modelos semânticos?

  - Métrica 1.3.a. $f(\text{TEMS}) \in \text{O}(|\text{M}|)$.

    Onde $|\text{M}|$ é a função de tamanho do modelo e $f(\text{TEMS})$ é a função do tempo de execução das extrações de modelos semânticos para todos os casos no experimento.

- Questão 1.4. Qual é o tempo necessário para aplicação de todas as regras de verificação para um modelo?

  - Métrica 1.4.a. TAC = $\sum_{1..n} \text{TR}_n$

    Onde TAC = Tempo total para aplicação das regras de computação e TR_n = Tempo de aplicação de uma enésima regra de computação.

- Questão 1.5. Qual é o tempo necessário para decidir se um modelo satisfaz a uma fórmula temporal?

  - Métrica 1.5.a. TV = TV_{final} - $\text{TV}_{inicial}$

    Onde TV = Tempo de Verificação, $\text{TV}_{inicial}$ = Tempo inicial antes da verificação e TV_{final} = Tempo final após a verificação. Dado um modelo M e uma fórmula Φ, TV é o tempo para decidir se M $\models \Phi$.

- Questão 1.6. Em que ordem de complexidade temporal está a função de execução da verificação?

  - Métrica 1.6.a. $f(\text{TV}) \in \text{O}(2^{|\Phi|})\text{O}(|\text{M}|)$.

    Onde $|\text{M}|$ é a função de tamanho do modelo, $|\Phi|$ é a função de tamanho da fórmula e $f(\text{TV})$ é a função do tempo de verificação para todos os modelos envolvidos no experimento. O algoritmo utilizado na verificação necessita satisfazer essa métrica.

- Questão 1.7. Qual é o tempo necessário para aplicação de verificações similares em um verificador de modelos de referência em desempenho temporal (Spin)?

– Métrica 1.7.a. TV_{SPIN} = $TV_{SPINfinal}$ - $TV_{SPINinicial}$
Onde TV_{SPIN} = Tempo de Verificação no Spin, $TV_{SPINinicial}$ = Tempo inicial antes da verificação na ferramenta Spin e TV_{final} = Tempo final após a verificação na ferramenta Spin. Dado um modelo M e uma fórmula Φ, TV é o tempo para decidir se M |= Φ.

Objetivo 2. Medir a capacidade da solução permitir verificação de sistemas de diferentes tamanhos.

- Propósito: medição;
- Questão: caracterização de capacidade espacial;
- Objeto: verificação de sistemas de diferentes tamanhos;
- Ponto de vista: provedor da solução.
- Questão 2.1. Qual é o número máximo de elementos de um modelo que a solução permite lidar?
  - Métrica 2.1.a. |M| = $\sum_{1..n}$ $Componente_n$. Onde |M| assume o valor máximo. A função de tamanho do modelo é a soma dos elementos de um modelo. Dessa forma, seleciona-se o valor máximo desse resultado.
- Questão 2.2. Em que ordem de complexidade espacial está a extração através das equações semânticas?
  - Métrica 2.2.a. $f(|MS|) \in O(|M|)$. Onde MS = Modelo semântico, |MS| = Tamanho do modelo semântico, e $f(|MS|)$ é a função de aplicação da extração a todos os modelos semânticos envolvidos no experimento.
- Questão 2.3. Qual é o custo que determina a complexidade espacial da aplicação da verificação?
  - Métrica 2.4.a. CV = $f(|M|,|\Phi|)$. Onde CV é o custo da verificação. Ela é uma função do tamanho do modelo em relação ao tamanho da fórmula.
- Questão 2.4. Em que ordem de complexidade espacial está o processamento das fórmulas de especificação?

– Métrica 2.3.a. $f(|M_\Phi|)\ in\ 2^{O(|\Phi|)}$.

Onde Φ é uma fórmula LTL, M_Φ é um modelo que representa Φ e $f(|M_\Phi|)$ é uma função do tamanho do modelo para todas as aplicações no experimento. Verificando-se a partir de cada sub-fórmula até a fórmula total, temos que esse conjunto de fórmulas pode ser armazenado em espaço polinomial. O algoritmo utilizado na verificação necessita satisfazer a essa métrica.

Objetivo 3. Detectar problemas de equivalência ao nível sintático entre conceitos transformados.

- Propósito: detecção;
- Questão: caracterização de falhas;
- Objeto: teste de equivalência sintática entre conceitos transformados;
- Ponto de vista: cliente MDA.
- Questão 3.1. A quais regras estruturais uma tabela sintática correta deve satisfazer?
  - Métrica 3.1.a. TES = $\bigcup_{1..n}$ Line$_n$.

    Onde TES = Tabela de equivalência sintática e Line$_n$ = linha de configuração para regras da transformação MDA.
  - Métrica 3.1.b. Line$_n$ = (Cell$_{in}$, Cell$_{out}$), com Cell$_{in}$ $\subseteq$ M$_{in}$.Componentes e Cell$_{out}$ $\subseteq$ M$_{out}$.Componentes.

    Onde Cell$_{in}$ = célula correspondente a uma linha na primeira parte da tupla e Cell$_{out}$ = célula correspondente a uma linha na segunda parte da tupla, M$_{in}$.Componentes = componentes do modelo de entrada e M$_{out}$.Componentes = componentes do modelo de saída.

Objetivo 4. Detectar falta de equivalência semântica entre conceitos transformados.

- Propósito: detecção;
- Questão: caracterização de problemas;
- Objeto: verificação de equivalência entre conceitos semânticos transformados;

- Ponto de vista: cliente MDA.

- Questão 4.1. Qual é a forma que as propriedades semânticas precisam satisfazer?

  - Métrica 4.1.a. MS = (S, I, R, L).

    Onde MS assume a forma de uma estrutura de Kripke. Tem S = conjunto finito de estados, I = conjunto de estados iniciais onde I $\subseteq$ S, R é uma relação de transição onde R $\subseteq$ S x S e L é uma função de nomes onde L : S -> 2^{AP}, com AP = proposições atômicas definidas para o modelo.

- Questão 4.2. Como apresenta-se uma diferença de propriedade semântica?

  - Métrica 4.2.a. Se (M_{in} |= Φ) $\neq$ (M_{out} |= Φ), temos falta de equivalência de uma propriedade semântica. Caso contrário, ainda é necessário verificar se CE(M_{in} |= Φ) $\cong$ CE(M_{out} |= Φ).

- Questão 4.3. Quais são as classes de propriedades semânticas existentes para a transformação analisada?

  - Métrica 4.3.a. Repete-se a verificação da métrica 4.2.a, considerando Φ como cada um dos casos a seguir. Para o caso específico desta avaliação, espera-se um alto grau de poder de detecção nas seguintes categorias de propriedades:

    * Propriedade Alcançabilidade (atingível): Φ = <>P.
    * Propriedade Segurança (nada de errado ocorrerá): Φ = [] P.
    * Propriedade Vivacidade (certeza de resposta): Φ = [](P -> <>Q).
    * Propriedade Equidade (justiça): Φ = []<>P.
    * Propriedade Liberdade de Deadlock (nunca trava): Φ = []X verdade.

  Finalmente, os padrões de especificação para lógica temporal apresentados no Capítulo 2.4.5 também deverão ser utilizados para essa questão.

- Questão 4.4. Qual é a eficácia da solução na captura de falhas em diversas situações de aplicação em modelos?

  - Métrica 4.4.a. Para o caso específico desta avaliação, espera-se que com variações exaustivas e empíricas dos modelos através da técnica de análise de mutantes, a

solução continue detectando erros de preservação entre modelos. A técnica adotada foi a Mutação Restrita [Mathur and Wong, 1994], baseando-se nos resultados de [Fabbri et al., 1995] para redes de Petri, onde alguns operadores de mutação foram selecionados para a geração dos mutantes. Desta forma, utilizamos a métrica escore de mutação, que é um valor no intervalo (0..1) e fornece uma medida da adequação do conjunto de testes analisados:

Dada a solução S e o conjunto de casos de testes T, calcula-se o escore de mutação ms(T,P) como ms(T,P) = DM(P,T) / (M(P) - EM(P)), onde:

DM(P,T): número de mutantes mortos pelo caso de teste T.

M(P): número total de mutantes gerados.

EM(P): número de mutantes equivalentes a P.

Desta forma, a equivalência entre mutantes a serem identificadas pelo experimento se dá pelo quão mais próximo o escore de mutação estiver de 1. Caso haja similaridade nos resultados do escore de mutação, inferimos que o operador de mutantes sob análise é preservador de semântica.

6.2 Planejamento

O estudo experimental baseia-se nas instanciações providas no Capítulo 4. Elas foram reusadas como guias para uma análise experimental de acordo com o cenário que construimos para uso do método GQM. Nesta seção, selecionamos o ambiente e os participantes, formulamos as hipóteses e fazemos uma preparação conceitual.

6.2.1 Seleção de Estudos de Caso

O mecanismo utilizado para realizar a avaliação foi por meio de vários experimentos controlados, com uma média de cinco execuções para cada cenário abordado, descritos logo a seguir. A aplicação dos estudos de caso escolhidos guiou-se pelos objetivos definidos anteriormente. Como cenário, reusou-se sistemas de automação e controle empregados pelo *cliente MDA* [Gomes and Costa, 2006], instanciado como o Projeto FORDESIGN, com sede na FCT (Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa). Esses sistemas foram especificados diretamente nas IOPTs ou através de diagramas comportamentais da UML abrangendo diversos níveis de complexidade e escalas. Entre eles, podemos citar, além de sistemas clássicos, também:

controlador de parque de estacionamento, controlador de tapetes rolantes em um sistema FIFO de produção industrial, controlador de processos paralelos sincronizados no início e no final da execução e controlador de edifícios inteligentes. Nos focamos nos modelos parque de estacionamento de 1 andar como descrito na Figura 51 e de 2 andares, não apresentado aqui por questões de complexidade deste modelo. Esses modelos estão envolvidos entre transformações envolvendo modelos PIM-para-PIM. Finalmente, como outro cenário de avaliação, experimentamos em transformações para sistemas domóticos, já em operação para condomínios residenciais, originalmente desenvolvidos em ANSI C para plataforma PIC e que foram re-modelados para uma arquitetura distribuída de acordo com as IOPTs e com sucessivas aplicações da transformação Splitting. O modelo utilizado na avaliação experimental está descrito na Figura 52 Assim, abordamos o contexto de transformações PSM-para-PSM, pois a origem e o destino estarão nas próprias IOPTs.

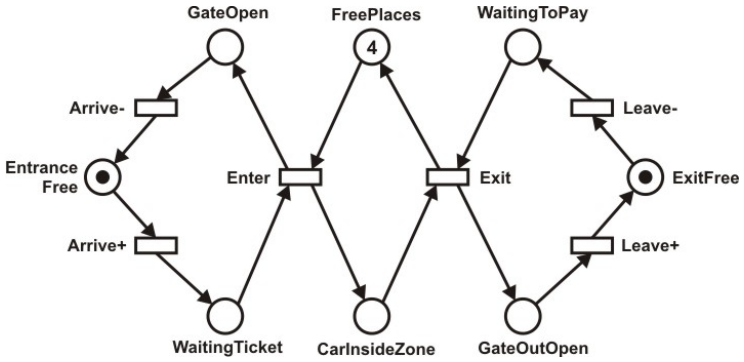

Figura 51: Modelo do Parque de Estacionamento de 1 andar empregado na avaliação experimental

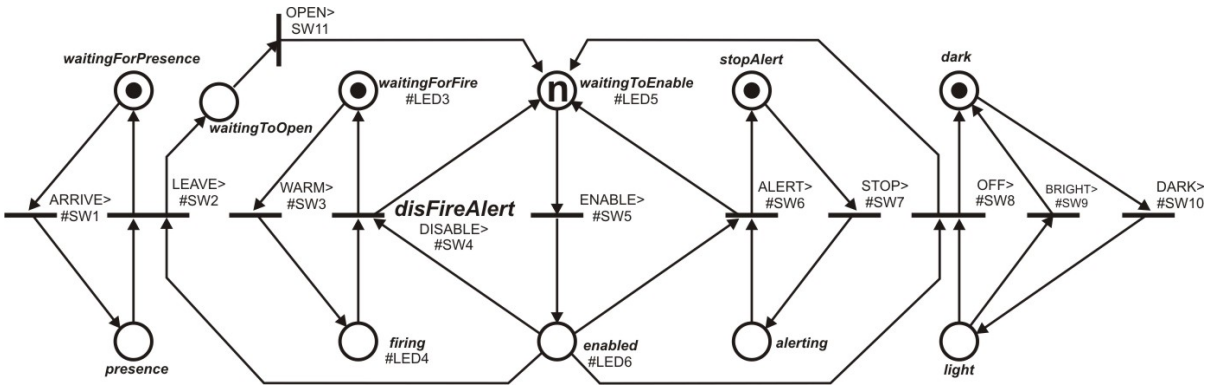

Figura 52: Modelo de condomínio residencial empregado na avaliação experimental

6.2.2 Seleção de Participantes

Os participantes selecionados produziram dados durante todo o ano de 2010. Durante esse período, houve uma interação direta com todos os atores que exercem os papéis que serão apresentados aqui.

A Figura 53 apresenta o fluxo geral de desenvolvimento em um contexto de interesse aos *clientes MDA*. Essa figura é importante para entendimento das atividades, dos atores e dos artefatos no contexto como um todo no qual essa avaliação experimental está inserida. Neste caso, analisaremos sob a ótica do papel de *Verificador dos Modelos*, que seria o papel com maior necessidade de especialização sob o ponto de vista de ser *cliente MDA*. Ele possui uma raia correspondente no diagrama apresentado na Figura 53. Porém, para que essa verificação possa dispor dos artefatos suficientes, o relacionamento com algumas outras raias precisa existir. Enumeramos, a seguir, os demais papéis existentes que tiveram algum envolvimento com esse trabalho de avaliação:

1. Modelador SOA: desenvolve atividade de modelagem de sistemas ao nível de construção de modelos CIM através das linguagens BPEL ou BPMN. Esse papel foi desempenhado por estudantes de iniciação científica da UFCG;

2. Modelador de Casos de Uso: desenvolve atividade de modelagem de sistemas ao nível de construção de modelos CIM através da linguagen UML. Esse papel não foi desempenhado em nossa avaliação experimental;

3. Modelador de IOPT: desenvolve atividades de modelagem e aplicação de transformações de sistemas ao nível de construção de modelos PIM/PSM através da extensão das redes de Petri denominadas por IOPTs. Esse papel foi desempenhado por pesquisadores da FCT;

4. Implantador de IOPT: aplica transformações de geração de código de mais baixo nível aos modelos IOPT visando execução em plataformas de sistemas embarcados. Esse papel foi desempenhado por estudantes de doutorado e mestrado da FCT e desenvolvedores de uma empresa especializada em desenvolvimento de sistemas embarcados.

5. Verificador de Modelos: especifica as propriedades a serem satisfeitas pelos modelos e aplica as verificações correspondentes. Esse papel foi desempenhado por estudantes de iniciação científica da UFCG e pesquisadores da FCT.

6. Integrador: Responsável pela interoperabilidade da solução com demais ferramentas e méto-

dos existentes no domínio de sistemas concorrentes. Embora desempenhe atividades essenciais ao projeto, não se requer informações desse ator para esse estudo experimental.

Ainda sobre a Figura 53, pode-se observar que o verificador dos modelos, que é o papel que conduz essa ótica da avaliação experimental, realiza tarefas desencadeadas pelo ator central do processo, que é o modelador IOPT, e que é responsável por construir modelos IOPT. O fato é que, mesmo que possamos trabalhar com diversos modelos por este processo, tais como modelos BPEL ou UML, ao final, todos serão convertidos em modelos IOPT, que é a linguagem formal adequada aos objetivos do *cliente MDA* e conseqüentemente aos nossos objetivos de validação. Ainda sobre a figura, podemos observar que a maneira de continuar o fluxo de desenvolvimento do modelador IOPT é através da aplicação da operação *Splitting*, que conforme já mencionada, é alvo principal da verificação dos seus modelos envolvidos como uma transformação MDA.

6.2.3 Configuração do Ambiente e Formulação das Hipóteses

Ambiente de Execução. Computadores Intel Dual Core 2.13 GHz com memória de 2 GB, ambiente Eclipse 3.3 Helios, Full Maude 2.5.

Formulação de Hipóteses. A seguir, apresentamos as hipóteses que permitirão avaliar a eficácia da solução para o problema abordado pelo trabalho:

1. Desempenho:

   (a) Participantes: modelador de IOPT e verificador de modelo.

   (b) Caracterização da hipótese: Para a transformação Splitting, o tempo de verificação na implementação existente dada pelo *cliente MDA* cresce de forma exponencial devido às características das abordagens para construção das regras de transformação. A aplicação das regras da implementação, conforme especificadas pelo *cliente MDA*, apresentarão tempos de ordem linear e logarítmica, a depender da característica de cada regra.

2. Escalabilidade:

   (a) Participantes: modelador de IOPT e verificador de modelo.

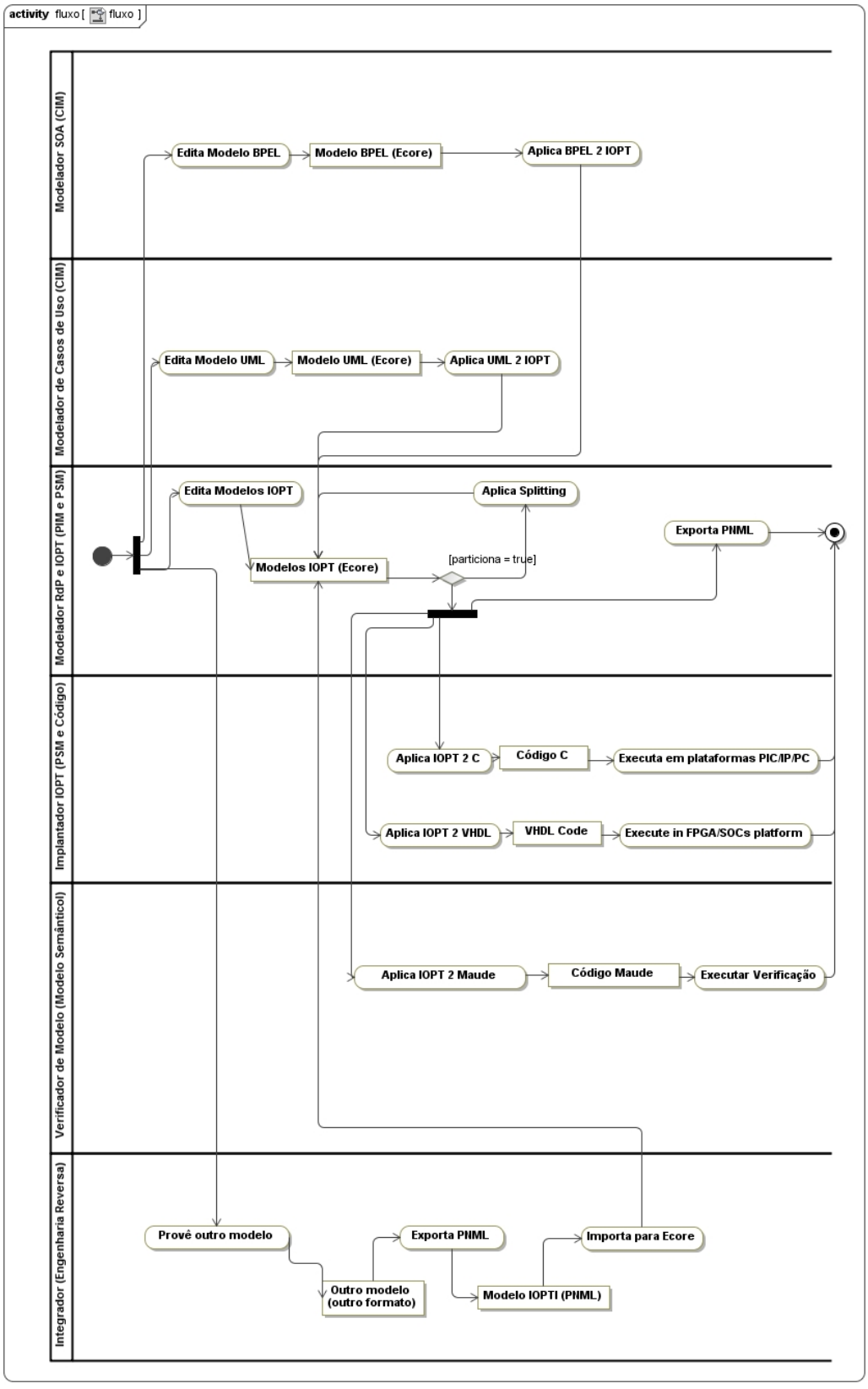

Figura 53: Atividades do projeto de acordo com os seus papéis

(b) Caracterização da hipótese: A solução permite lidar, em termos de escala, com os modelos apresentados nos estudos de caso, que apresentam tamanhos variados e diferentes níveis de complexidade.

3. Problemas Sintáticos Encontrados:

   (a) Participantes: modelador de IOPT e verificador de modelo.

   (b) Caracterização da hipótese: A técnica permite uma descoberta sintática por uma tabela de equivalência dos conceitos erroneamente representados no modelo de saída e uma classificação clara desses tipos de defeitos. Além disso, deverá ser possível a detecção de falhas na boa-formação dos modelos em relação à sua especificação.

4. Problemas Semânticos Encontrados:

   (a) Participantes: modelador de IOPT e verificador de modelo.

   (b) Caracterização da hipótese: A técnica permite uma descoberta de erros semânticos por meio da aplicação de verificação de modelos, detectando comportamentos inesperados no modelo de saída. Com a descoberta dessas faltas de equivalência, contra-exemplos que permitam o entendimento da característica comportamental falha deverão se apresentados também.

6.2.4 Preâmbulo: Avaliação do cenário

A solução MDA-Veritas possui uma implementação como protótipo que possibilitou implementarmos o estudo experimental, possibilitando a construção dos estudos de caso selecionados. A ferramenta possui demonstrações e artefatos disponíveis em [MDA-VERITAS, 2011] para download. Os casos de uso dessa ferramenta, também apresentados na Figura 54, que são interessantes para nossa avaliação estão resumidos a seguir:

1. Criar, editar, carregar e salvar um modelo rede de Petri ou IOPT - O sistema utiliza arquivos providos pelo PNML Framework, provê uma forma de converter versões mais antigas do formato PNML na atual, provê a opção de carregar arquivos IOPT criados pelo editor da linguagem e também possibilita salvar as redes de Petri em um formato que possa ser lido por esse mesmo editor.

2. Executar a operação Splitting - A entrada da transformação é o modelo em rede de Petri ou IOPT a ser particionado de acordo com o metamodelo PNML e uma lista de lugares e/ou transições (conjunto de corte). Há as seguintes restrições:

   (a) Ambas as redes e o arquivo de parâmetro devem estar de acordo com o metamodelo PNML;

   (b) O sistema deve verificar se o conjunto de corte é válido, em prioridade com a execução da transformação Splitting;

   (c) O sistema deve salvar a nova rede particionada em um novo arquivo, escolhido pelo usuário;

   (d) O usuário deve escolher se a rede particionada será carregada em memória após a transformação ou não.

3. Gerar o código de verificação - A ferramenta escolhida como formato de saída das equações semântica foi o Maude. Temos os seguintes casos de uso para essa parte:

   (a) O sistema deve prover uma área de texto aonde o usuário possa submeter como entrada as fórmulas de verificação desejadas;

   (b) A boa formação de fórmulas é checada pelo Maude e se existir alguma má formação, será exibida pelo console do Eclipse.

4. Executar o código de verificação:

   (a) O sistema deve permitir que o usuário execute o código de verificação e veja os resultados da verificação no visor do console do Eclipse;

   (b) Algum mecanismo deverá ajudar ao usuário na interpretação dos resultados.

Os modelos foram criados utilizando o Taipa, um editor de redes de Petri provido pelo PNML Framework. Taipa é desenvolvido junto com o Eclipse Graphical Framework (GMF), uma ferramenta que objetiva uma prototipagem rápida de editores de linguagens específicas de domínio. Um modelo criado em Taipa permite uma visão consistente da representação visual da rede de Petri e do modelo Ecore que utiliza-se nas transformações.

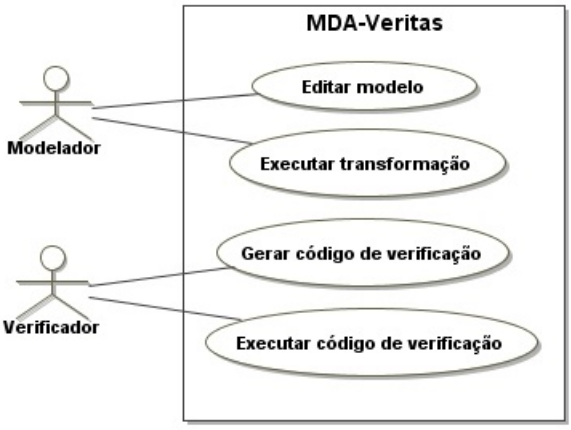

Figura 54: Casos de uso da versão ferramental da MDA-Veritas

6.2.5 Seleção de Variáveis

Variáveis Independentes:

1. A transformação Splitting (Tr), exercendo o papel de PIM-para-PIM ou de PSM-para-PSM;
2. O modelo de entrada da transformação (M_{in}), descrito em redes de Petri ou em redes IOPT;
3. O tamanho em elementos de rede do modelo de entrada da transformação ($|M_{in}|$);
4. O modelo de saída da da transformação (M_{out}), descrito em redes de Petri ou em redes IOPT;
5. O tamanho do em elementos de rede do modelo de saída da transformação ($|M_{out}|$);
6. A tabela de equivalência sintática (TES);
7. As equações semânticas do modelo de entrada (EqS_{in}), descritas em ATL;
8. As equações semânticas do modelo de saída (EqS_{out}), descritas em ATL;
9. O verificador formal ($\cong$), descrito como o model-checker Maude ou o model-checker Spin;
10. As especificações das propriedades a serem verificadas (P);
11. O número de regras aplicadas em uma verificação (RC_{in} e RC_{out});
12. O tamanho da fórmula de especificação em termos de proposições e operadores ($|\Phi|$);
13. Os padrões de especificação das propriedades semânticas (P).

Variáveis Dependentes:

1. O tempo total de aplicação da transformação (TT);
2. O tempo total de aplicação da verificação (TV);
3. O resultado booleano da verificação de um modelo (RM = (M |= Φ));
4. O caminho de contra-exemplo provido pela verificação de um modelo (CE);
5. O espaço de estados do modelo rede de Petri de entrada da transformação (MS_{in});
6. O espaço de estados do modelo IOPT de saída da transformação (MS_{out});
7. O resultado da verificação de equivalência na comparação dos resultados de verificação dos dois modelos envolvidos na transformação (RV = ((M_{in} |= Φ) = (M_{out} |= Φ) ou (M_{in} |= Φ) $\neq$ (M_{out} |= Φ)));
8. O escore de mutação, conforme apresentado na Métrica 4.4.a (ms(T,P)).

Ainda sobre o escore de mutação, devido à influência que a transformação MDA estudada exerce sobre os modelos, temos que ele foi analisado em termos dos seguintes operadores de mutação:

1. InitMarkDel, que faz remoção da marcação inicial dos modelos;
2. InitMarkIns, que faz inserção de uma nova marcação inicial nos modelos;
3. InitMarkExchange, que faz a troca da marcação inicial nos modelos;
4. InputArcInversion/OutputArcInversion, que fazem inversão de arcos de entrada e de saída nos modelos;
5. InputArcDel, que faz remoção de arco de entrada nos modelos;
6. InputArcExchange, que troca de arco de entrada nos modelos;
7. OutputArcExchange, que troca de arco de saída nos modelos.

6.3 Execução e Resultados

Nesta seção, coletamos os dados necessários para o cálculo das métricas que serão analisadas na fase posterior de interpretação. A seguir, detalhamos a definição das execuções e os resultados obtidos com os estudos de caso apresentados. Como ilustração, este documento apresenta todo o procedimento para alguns modelos selecionados.

Iniciamos abordando o artefato transformação. A Operação Splitting foi implementada como uma transformação entre modelos na linguagem de transformação ATL. A transformação como um todo foi dividida em três passos:

1. Remoção do conjunto de corte;
2. Checagem de conexão (divisão de sub-redes);
3. Particionamento.

A busca pelo tempo de execução desses procedimentos corresponde procurar responder à Questão 1.1. Implementamos ambos os itens 1 e 3 em ATL e o item 2 em Java. Isto ocorreu devido ao fato de que foi necessária a implementação de um algoritmo de busca em amplitude para verificar se a remoção do conjunto de corte gerou duas ou mais redes. Tentou-se uma implementação desse algoritmo de busca em amplitude também em ATL, mas verificou-se problema de escalabilidade para modelos maiores como será descrito mais adiante. Isto foi considerado como o principal gargalo da implementação da transformação, e sua implementação em Java fez com que o espaço em memória para execução desse passo fosse diminuído significativamente.

Classificamos a execução da transformação de uma maneira polimórfica. Se o modelo de entrada e saída for uma rede de Petri, teremos modelos PIMs envolvidos. Se o modelo de entrada e saída for uma rede IOPT, teremos modelos PSMs envolvidos. Dessa forma, executamos verificação para estudos de caso específicos em dois tipos de transformações: (A) PIM-para-PIM; e (B) PSM-para-PSM. Os resultados para cada um dos casos estão detalhados nas seções a seguir.

6.3.1 Resultados em Verificação de Modelos em uma Transformação PIM-para-PIM

O exemplo que guia a avaliação da técnica em transformações PIM-para-PIM é um controlador de estacionamento simples, conforme já foi descrito no Capítulo 5.1.1. Outros modelos sob a característica de transformações PIM-para-PIM também receberam avaliação. Dados de experimentos para demais modelos podem ser encontrados no site do projeto [MDA-VERITAS, 2011].

Basicamente, temos a necessidade de dois controladores independentes nesse sistema: um para a entrada e outro para a saída de automóveis. Para efeitos de análise da complexidade da técnica, também consideraremos um modelo com 2 andares de estacionamento, que é um modelo que incrementa complexidade do parque de 1 andar em aproximadamente quatro vezes o número de elementos de redes de Petri.

Partindo dos dois modelos PIMs, visando responder à Questão 1.1, e utilizando as métricas 1.1.a e 1.1.b, obtemos as execuções apresentadas na Tabela 11.

Execuções	**TT1**	**TTS1**	**TT2**	**TTS2**
1	1.435 s	0.451 s	14.56 s	1.050 s
2	1.714 s	0.421 s	13.21 s	1.391 s
3	1.180 s	0.393 s	12.18 s	0.869 s
4	1.218 s	0.417 s	12.45 s	1.122 s
5	0.987 s	0.399 s	12.91 s	1.209 s
Média	1.306 s	0.416 s	13.06 s	1.041 s

Tabela 11: Execução da transformação Splitting para modelos do Parque de 1 e 2 andares

A média total de aplicação da transformação para o parque de 1 andar, nomeada TT1, é de 1.306 segundos. Este tempo foi considerado adequado pelo *cliente MDA* para execução de transformações. A aplicação da extração da tabela sintática TTS1 ficou em 0.416 segundos. Isto significa dizer que esta aplicação, no processo da transformação, representa 31% do tempo total de aplicação do processo. Como essa técnica pode ser aplicada concorrentemente à transformação, *a extração da tabela sintática não acrescentou tempo significativo ao tempo de aplicação da transformação*.

A média total da aplicação da transformação para o parque de 2 andares, nomeada TT2, foi de 13.06 segundos. Isto caracteriza uma situação de aplicação da transformação em que a complexidade maior dos modelos influenciou proporcionalmente em seu desempenho. Porém, o tempo de extração da tabela sintática, TTS2, foi de 1.041 segundos, que é um percentual ainda abaixo do obtido anteriormente, viabilizando o uso da nossa técnica.

Concluímos que o tempo de construção da tabela sintática não interfere no tempo de aplicação da transformação.

A ameaça à validade desse experimento reside em algum possível caso em que o tempo de verificação do conjunto de corte cresça exponencialmente. Nesse caso, seria necessário varrer

todos os elementos que compõem o modelo. Contudo, esse caso não foi abordado na formalização original da transformação.

Para começarmos a responder à Questão 1.2 e à Questão 1.3, através do uso das métricas 1.2.a e 1.3.a, tomamos o tempo de extração dos modelos TEMS através das equações semânticas. Na Tabela 12, consideramos detalhes de tempo para a transformação do modelo semântico e da geração de código para o modelo de entrada (M_{in}) e resumimos o TEMS para o modelo de saída (M_{out}):

Execuções	**Transf para o modelo semântico**$_{in}$	**Geração de código**$_{in}$	**TEMS**$_{in}$	**TEMS**$_{out}$
1	1.316 s	0.398 s	1.714 s	1.728 s
2	1.085 s	0.365 s	1.450 s	1.557 s
3	1.118 s	0.344 s	1.462 s	1.560 s
4	1.224 s	0.408 s	1.632 s	1.621 s
5	1.098 s	0.438 s	1.536 s	1.601 s
Média	1.168 s	0.390 s	1.558 s	1.613 s

Tabela 12: Extração das equações semânticas para modelos do Parque de 1 andar

Baseado no parâmetro TEMS da Tabela 12, observamos na Tabela 13 o comportamento da extração do modelo semântico para um modelo considerado de uma ordem bem superior em tamanho. Como pode-se observar, o tempo de extração não aumentou na mesma proporção em que a complexidade do modelo evoluiu. Dessa forma, temos viabilizado o uso das *equações semânticas*.

Execuções	**Transf para o modelo semântico**$_{in}$	**Geração de código**$_{in}$	**TEMS**$_{in}$	**TEMS**$_{out}$
1	1.411 s	0.406 s	1.817 s	1.886 s
2	1.602 s	0.522 s	2.124 s	1.929 s
3	1.412 s	0.428 s	1.840 s	1.972 s
4	1.419 s	0.432 s	1.851 s	2.002 s
5	1.573 s	0.403 s	1.976 s	2.139 s
Média	1.483 s	0.438 s	1.921 s	1.985 s

Tabela 13: Extração das equações semânticas para modelos do Parque de 2 andares

Conluímos que o tempo de extração do modelo semântico não foi afetado pela evolução da complexidade dos modelos. Além do mais, o tempo de extração do modelo semântico tem limite superior definido pela evolução de complexidade dos modelos.

A ameaça à validade desse experimento reside no uso de algum domínio semântico complexo o suficiente para que sua tradução ultrapasse a complexidade dos modelos. Contudo, isso não se verifica nos modelos semânticos adotados, servindo de alerta apenas para outros casos futuros de instanciações da MDA-Veritas.

Para buscarmos interpretações às Questões 1.4, 1.5 e 1.6, especificamos propriedades utilizando LTL para serem verificadas no Maude model-checker. Empregamos todo o poder de expressividade dos padrões de especificação apresentados na Seção 6.1. As propriedades foram sistematicamente escolhidas de acordo com a necessidade do ator *modelador de IOPTs*. A Tabela 14 apresenta os dados para verificação de *deadlock* em ambos modelos envolvidos na transformação para o parque de 1 e 2 andares.

Execuções	**TV1**$_{in}$	**TV1**$_{out}$	**TV2**$_{in}$	**TV2**$_{out}$
1	0.005 s	0.007 s	7.395 s	9.219 s
2	0.006 s	0.007 s	7.216 s	8.937 s
3	0.005 s	0.008 s	7.905 s	9.084 s
4	0.006 s	0.007 s	8.092 s	9.107 s
5	0.005 s	0.007 s	7.884 s	9.371 s
Média	0.0056 s	0.0072 s	7.480 s	9.147 s

Tabela 14: Verificação de deadlock nos dois modelos parque de 1 e 2 andares

Percebe-se que a análise no modelo do parque em 2 andares, nomeada por TV2, apresenta um grande incremento de complexidade temporal em relação aos modelos de 1 andar, nomeado por TV1. Percebe-se claramente na Tabela 15, que os modelos de saída da transformação sofrem de um pequeno incremento em seu tempo de verificação em relação aos modelos de saída, embora não haja uma diferença da mesma ordem em variação no seu espaço de estados e por possuirem a mesma fórmula de verificação para a entrada. Isso ocorre devido ao fato do modelo de saída estar representado no FULL-Maude, para que seja possível expressar a modularidade idealizada pela transformação Splitting. Contudo, isto exige representação de regras de computação mais complexas. Para reforçar essa observação, a tabela ainda apresenta os dados em função de tempo de verificação e do número de reescritas (NR) para os modelos do parque de 1 andar durante a busca por deadlock, onde fazemos uso da métrica 1.4.a, derivando, dessa forma, o tempo médio de reescrita (TR) para um dado modelo. Ainda para essas tabelas, considera-se o tempo total de verificação (TV) equivalente ao tempo total de aplicação das regras de computação (TAC) por ser

o comando utilizado para construção dos espaços de estados dos sistemas.

Execuções	**TV2**$_{in}$	**NR**$_{in}$	**TR**$_{in}$	**TV2**$_{out}$	**NR**$_{out}$	**TR**$_{out}$
1	7.395 s	54	0.136 s	9.219 s	35	0.263 s
2	7.216 s	54	0.133 s	8.937 s	35	0.255 s
3	7.905 s	54	0.146 s	9.084 s	35	0.259 s
4	8.092 s	54	0.149 s	9.107 s	35	0.260 s
5	7.884 s	54	0.146 s	9.371 s	35	0.267 s
Média	7.480 s	54	0.142 s	9.147 s	35	0.260 s

Tabela 15: Verificação da propriedade de ausência de deadlock nos dois modelos PIM da transformação para o parque de 2 andares

Observamos que a média do custo de uma regra de computação (TR) para M$_{out}$ (0.260 s) tornou-se bem mais cara quando comparado com a média do custo para M$_{in}$ (0.142 s).

Procedemos com outras verificações, apresentado-as de uma maneira mais simplificada, utilizando fórmulas de verificação LTL a nível de complexidade incremental, mas que teremos o mesmo padrão preservado. As fórmulas, que serão bastante empregadas adiante, são referenciadas de acordo com a seguinte enumeração:

1. `red modelCheck(initial, [] enabled ).` Objetiva buscar pela propriedade de vivacidade. Objetivamos saber se haverá uma transição habilitada para todos os estados do espaço de estados.

2. `red modelCheck(initial, ~ <> (Enter /\ O(Arrive- /\ O (Arrive+ /\ O (Leave+ /\ O Exit)))))` para M$_{in}$
   e
   `red modelCheck(initial, ~ <> (Enter;Enter\_copy /\ O(Arrive- /\ O (Arrive+;Arrive+\_copy /\ O (Leave+ /\ O Exit)))))` para M$_{out}$.
   Objetiva buscar por caminhos de eventos específicos em ambas as redes. As propriedades variam apenas pela especificação do caminho, pois deve haver uma consulta à tabela de equivalência sintática.

3. `[](Enter -> <>(Exit)).` Significa o uso do padrão Universalidade de P após Q. Nesse caso, estamos interessados em saber se sempre que o portão for aberto para um carro, em algum momento esse carro obrigatoriamente irá sair.

Nas Tabelas 16 e 17, ilustramos os dados para o desempenho desses três tipos de especificações de propriedades nos modelos do parque de 1 andar e de 2 andares. Cada item enumerado acima terá dois campos na tabela: um para o tempo de verificação da fórmula (TV) em M_{in} e outro para a fórmula em M_{out}. Assim, por exemplo, o campo 1-M_{in} refere-se à verificação da propriedade 1 enumerada acima para o modelo M_{in}. A sintaxe é *número-modelo*.

Execuções/TV	**1-M$_{in}$**	**1-M$_{out}$**	**2-M$_{in}$**	**2-M$_{out}$**	**3-M$_{in}$**	**3-M$_{out}$**
1	0.005 s	0.009 s	0.003 s	0.005	0.008 s	0.015 s
2	0.006 s	0.009 s	0.004 s	0.005	0.009 s	0.013 s
3	0.005 s	0.009 s	0.002 s	0.004	0.010 s	0.013 s
4	0.005 s	0.009 s	0.003 s	0.005	0.010 s	0.013 s
5	0.005 s	0.009 s	0.003 s	0.005	0.009 s	0.013 s
Média	0.0052 s	0.009 s	0.003 s	0.0048	0.0093 s	0.0135 s

Tabela 16: Verificação das 3 propriedades para modelos do Parque de 1 andar

Execuções/TV	**1-M$_{in}$**	**1-M$_{out}$**	**2-M$_{in}$**	**2-M$_{out}$**	**3-M$_{in}$**	**3-M$_{out}$**
1	7.782 s	8.441 s	7.243 s	18.701	9.201 s	24 s
2	7.688 s	8.349 s	7.040 s	20.111	9.043 s	29 s
3	7.869. s	8.911 s	7.802 s	24.714	9.931 s	27 s
4	7.190 s	8.988 s	8.003 s	20.891	9.472 s	27 s
5	6.950 s	8.592 s	8.464 s	21.900	9.847 s	28 s
Média	7.48 s	8.64 s	7.68 s	21.24	9.498 s	27 s

Tabela 17: Verificação das 3 propriedades para modelos do Parque de 2 andares

Percebemos que o custo da regra de computação fica cada vez mais elevado nos modelos distribuídos à medida que a complexidade na computação da distribuição dos componentes dos modelos aumenta. O aumento da diferença de tempo de verificação entre M_{in} e M_{out} indica esse fator.

Concluímos que o custo da regra de computação está diretamente associada à complexidade do modelo.

A ameaça à validade desse experimento reside em casos de variação do custo de computação de uma regra em particular que possa alterar de maneira determinante o custo médio calculado.

A interpretação dada a seguir, dada sobre a métrica 1.5.a, sugere que conseguimos lidar satisfatoriamente em termos temporais com o problema de model-checking para nossa solução. Isto

significa dizer que a aplicação de algoritmos de verificação não atrapalha o tempo a ser utilizado nas demais atividades de desenvolvimento. O mesmo vale para a métrica 1.6.a, onde tivemos que as fórmulas de verificação foram decididas em tempo satisfatório através da capacidade do Maude LTL model-checker.

Concluímos que o tempo de verificação da propriedade de um modelo é dependente da ordem de prioridade do processamento da regra de computação, do tamanho da fórmula e do espaço de estados respectivamente.

A ameaça à validade desse experimento reside no problema da explosão do espaço de estados, comumente encontrado em problemas de verificação. Nesse caso, não seria possível caracterizar tempo de verificação.

Finalmente, para fechar o objetivo de medição de desempenho, vamos à Questão 1.7, onde observamos como o model-checker Spin se comporta diante das mesmas condições de verificação para uma dada forma de representar modelos em redes de Petri. Não nos interessamos pela representação de um modelo particionado (M_{out}) nessa ferramenta por questões de escopo do trabalho. Os dados das Tabelas 18 e 19 comparam os tempos de verificação para as 3 propriedades enumeradas anteriormente com o Maude e com o Spin para o modelo de entrada (M_{in}). Por exemplo, temos que 1-Maude, refere-se ao tempo de verificação da propriedade 1 na ferramenta Maude. A sintaxe é *número da propriedade - ferramenta*.

Execuções	**1-Maude**	**1-Spin**	**2-Maude**	**2-Spin**	**3-Maude**	**3-Spin**
1	0.005 s	0.004 s	0.003 s	0.002	0.008 s	0.003
2	0.006 s	0.004 s	0.004 s	0.002	0.009 s	0.003
3	0.005 s	0.004 s	0.002 s	0.002	0.010 s	0.003
4	0.005 s	0.004 s	0.003 s	0.002	0.010 s	0.003
5	0.005 s	0.004 s	0.003 s	0.002	0.009 s	0.003
Média	0.0052 s	0.004 s	0.003 s	0.002	0.0093 s	0.0135

Tabela 18: Comparação de tempo de verificação Maude x Spin para modelos do Parque de 1 andar

Observamos que em alguns casos, principalmente quando a complexidade dos modelos aumenta, que o model-checker Spin consegue resolver o problema de verificação da propriedade (TV_{SPIN}) em até 80% do tempo do Maude. Para o projeto em questão, esses dados não foram significativos, haja visto que na contra-balança estão os inúmeros benefícios, já discutidos anteriormente, que a semântica translacional para o Maude nos trouxe.

Execuções	1-Maude	1-Spin	2-Maude	2-Spin	3-Maude	3-Spin
1	7.782 s	5.381 s	7.243 s	6.220	9.201 s	7.801 s
2	7.688 s	5.622 s	7.040 s	6.100.	9.043 s	7.772 s
3	7.869. s	5.021 s	7.802 s	5.977	9.931 s	7.439 s
4	7.190 s	5.783 s	8.003 s	5.928	9.472 s	7.943 s
5	6.950 s	5.922 s	8.464 s	6.002	9.847 s	7.700 s
Média	7.48 s	8.64 s	7.68 s	6.044	9.498 s	7.731 s

Tabela 19: Comparação de tempo de verificação Maude x Spin para modelos do Parque de 2 andares

A verificação das propriedades no Maude apresenta tempos aceitáveis ao compararmos com a principal referência em eficiência em model-checking, que é o model-checker Spin.

A ameaça à validade desse experimento reside novamente no problema da explosão de estados. Não sabemos qual seria o limite dessas duas ferramentas. Em caso do limite do Maude ser bem inferior ao Spin, alguns experimentos não poderiam ser repetidos.

Para trabalharmos sobre o objetivo de *escalabilidade*, necessitamos proceder igualmente à questão do desempenho, montando uma tabela de avaliação e averiguando os dados. A variável tempo ainda estará presente, porém serve apenas para identificar quando houve explosão de estados. Vejamos na Tabela 20, que relaciona tempos de verificação (TV), número de reescritas (NR) e tamanho dos modelos (|M|), como a solução se comporta quando escalamos o número de lugares para estacionar, para o modelo parque de 1 andar. Isso nos permite fazer interpretações sobre as Questões 2.1, 2.2, 2.3 e 2.4.

Observamos que, surpreendentemente, no contexto de exercício de verificação com modelos de ordem maior, o modelo particionado (M_{out}), que é a saída da transformação Splitting, escala bem além do modelo centralizado (M_{in}), que corresponde à entrada da transformação Splitting. Ainda há de se observar que este resultado segue essa tendência mesmo existindo um número maior de estados e de reescritas M_{out} devido às mensagens de comunicação entre os módulos. Isto se deve à maior eficiência provida pelo FULL-Maude, módulo que permite a representação de modelos particionados. Este resultado, embora diferente, é de enorme utilidade para a nossa técnica, pois como modelos PIMs possuem nível de abstração maior, dificilmente eles serão tão complexos quanto os modelos PSMs que necessitam representar conceitos específicos de plataformas. Dessa forma, a representação em PSMs foi realizada em FULL-Maude, tanto por poderes de expressão da linguagem, como também, conforme demonstrado, por questões de eficiência.

Marcas	**TV**$_{in}$	**NR**$_{in}$	**|M**$_{in}$**|**	**TV**$_{out}$	**NR**$_{out}$	**|M**$_{out}$**|**
1	0.1 s	30	18	0.1 s	53	30
2	0.1 s	48	27	0.1 s	83	45
3	0.1 s	66	36	0.1 s	113	60
4	0.1 s	84	45	0.1 s	143	75
5	0.2 s	102	54	0.2 s	173	90
7	0.2 s	138	72	0.2 s	233	120
10	0.2 s	192	99	0.2 s	323	165
15	0.3 s	282	144	0.3 s	473	240
20	0.4 s	372	189	0.3 s	623	315
25	0.5 s	462	234	0.3 s	773	390
30	0.6 s	552	279	0.3 s	923	465
35	0.7 s	642	324	0.3 s	1073	540
40	0.8 s	732	369	0.4 s	1223	615
50	0.8 s	912	459	0.5 s	1523	765
75	1 s	1362	684	0.6 s	2273	1140
100	1 s	3612	909	0.8 s	3023	1515
200	1 s	4712	1809	1 s	6023	3015
300	1 s	5412	2709	1 s	9023	4515
400	1 s	7212	3609	1 s	12023	6015
500	1 s	9012	4509	1 s	15023	7515
700	1 s	12612	6309	1 s	21023	10515
1000	2 s	18012	9009	1 s	30023	15015
1500	8 s	27012	13509	1 s	45023	22515
2000	21 s	36012	18009	2 s	60023	30015
2500	46 s	45012	22509	4 s	75023	37515
3000	77 s	54012	27009	11 s	90023	45015
3500	164 s	63012	31509	14 s	105023	52515
4000	250 s	72012	36009	22 s	120023	60015
5000	586 s	90012	45009	51 s	150023	75015
10000	10386 s	180012	90009	258 s	300023	150015
20000	-	-	-	687 s	600023	300015
30000	-	-	-	1667 s	900023	450015

Tabela 20: Escalabilidade de verificação para os modelos de entrada e saída do parque de 1 andar

Concluímos que pudemos lidar com modelos PIM com número de elementos de redes de Petri que satisfizeram ao cliente MDA. Além do mais, a ordem do custo espacial para extração de modelos semânticos está de acordo com a evolução do tamanho do modelo.

A ameaça à validade desse experimento reside novamente no problema da explosão de estados.

A próxima questão pôde ser interpretada observando-se a Tabela 21 para os dois modelos (M_{in} e M_{out}) e inferindo até onde a solução escalou. Aplicando a verificação de outras propriedades especificadas em LTL aos modelos, interpretamos de acordo com o número de estados, da fórmula de especificação e das reescritas para cada possibilidade.

Concluímos que a solução FULL-Maude, que usamos para representar modelos particionados, demonstrou ser bastante poderosa no quesito escalabilidade. O consumo aumentou de acordo com a evolução do modelo verificado e da fórmula empregada na especificação.

A ameaça à validade desse experimento reside além do problema da explosão de estados, na pragmática do FULL-Maude em situações que não seja adequado o uso dessa ferramenta. Até então, não conseguimos identificar quais seriam essas situações particulares.

Observamos pela Tabela 23, que existe aparentemente alguma razão matemática no crescimento do número de reescrita à medida em que o número de marcas representando lugares de estacionamento aumenta. Um fator limitante para a solução é o hardware utilizado para a computação. Para verificações em que os resultados são obtidos diretamente (propriedade 3), não observamos nenhuma limitação. Dessa forma, podemos formular nossa última interpretação para a última questão de escalabilidade.

Concluímos que o problema da verificação esteve em tempo polinomial, conforme sugerido pela teoria da técnica de model-checking. Os limites de aplicação se deram por ineficiência do hardware utilizado.

A ameaça à validade desse experimento reside novamente no problema da explosão de estados.

A busca por interpretações à Questão 3.1, do objetivo de *busca por equivalência sintática*, estão apresentadas nos tópicos a seguir e foram obtidas de forma automática, semi-automática e por interações com os atores envolvidos nesse processo. A primeira categoria apresenta casos em que houve claramente uma falta de entendimento da especificação por parte do implementador da transformação, de acordo com as métricas 3.1.a e 3.1.b por envolver diretamente os campos da *tabela sintática*, instanciadas em termos da transformação Splitting:

1. Não percebeu-se que a remoção precisa gerar redes que estejam desconexas. Nenhum nó

Marcas	**1-M**$_{in}$	**1-M**$_{out}$	**2-M**$_{in}$	**2-M**$_{out}$	**3-M**$_{in}$	**3-M**$_{out}$
1	53	87	43	83	8	8
2	80	132	50	116	8	8
3	107	177	53	149	8	8
4	134	222	56	182	8	8
5	161	267	59	215	8	8
7	215	357	65	281	8	8
10	296	492	74	380	8	8
15	431	717	89	545	8	8
20	566	942	104	710	8	8
25	701	1167	119	875	8	8
30	836	1392	134	1040	8	8
35	971	1617	149	1205	8	8
40	1106	1842	164	1370	8	8
45	1241	2067	179	1535	8	8
50	1376	2292	194	1700	8	8
70	1916	3192	254	2360	8	8
100	2726	4542	344	3350	8	8
200	5426	9042	644	6650	8	8
300	8126	13542	944	9950	8	8
400	10826	18042	1244	13250	8	8
500	13526	22542	1544	16550	8	8
700	18926	31542	2144	23150	8	8
1000	27026	45042	3044	33050	8	8
1500	40526	67542	4544	49550	8	8
2000	54026	90042	6044	66050	8	8
2500	67526	112542	7544	82550	8	8
3000	81026	135042	9044	99050	8	8
3500	94526	157542	10544	115550	8	8
4000	-	180042	12044	132050	8	8
4500	-	202542	13544	148550	8	8
5000	-	225042	15044	165050	8	8
10000	-	450042	30044	330050	8	8
20000	-	-	-	-	8	8
30000	-	-	-	-	8	8

Tabela 21: Verificação das 3 propriedades observando-se escala para o Parque de 1 andar

envolvido em um conflito estrutural ficou em sub-redes diferentes.

2. Falta de definição de operação de retirada dos nós. Não percebeu-se, em nenhum momento, que essa operação também seria um componente essencial da transformação como um todo.

3. Falta de verificação do conjunto de corte. A transformação só poderia ser efetuada corretamente se partisse de um conjunto de corte válido.

4. As operações de pré-condição com as regras de aplicação geraram alguns maus entendimentos. Um exemplo disso foi a escolha errônea em diversas situações dos elementos de entrada ou de saída de um determinado nó da rede que foi anexado ao conjunto de corte.

5. Falta de verificação para a questão de se todos os elementos de entrada de um nó estão no componente atual. O procedimento de transformação da Splitting requisita uma verificação final dos elementos de entrada dos componentes particionados.

6. Problema na implementação do algoritmo de verificação de conectividade. Este algoritmo achou um número diferente do esperado de sub-redes.

7. Desaparecimento de nós na rede original que foram envolvidos no conjunto de corte.

8. Erros originados da remoção de elementos de entrada e de saída de determinados nós que pertenceriam ao conjunto de corte.

9. Alguns erros de divisão ainda foram causados por más interpretações dos conflitos.

10. Criação de transições escravas desnecessárias.

11. Erro de conjunto de corte inválido. Esse erro repetiu-se desde uma versão anterior não implementada com linguagens MDA.

12. Falha na detecção de qual deve ser o componente mestre da transformação.

13. Resultados inesperados com mudança de opção para o componente mestre.

14. Causas sintáticas relacionadas à má interpretação da especificação. Um dos principais exemplos foram questões relacionadas à condição de parada quando percorrendo os modelos envolvidos nas transformações.

15. Checagem de conectividade esteve ineficiente por características da linguagem de transformação. Infelizmente, a linguagem de transformação que utilizamos, ou o seu manuseio, não correspondeu a certas expectativas para que utilizássemos algumas construções sintáticas. O uso dessas construções implicava em uma ineficiência direta da transformação.

16. Algoritmos que executam para cada nó do grafo baseando-se em contador. Esta maneira exige um número de visitas muito alto a cada elemento que compõe a rede, levando a diversos problemas de eficiência.

Há muitas dependências para a execução da transformação analisada, devido ao seu caráter complexo. Estes resultados inesperados ocorreram devido às limitações ainda existentes no mecanismo de especificação e implementação da transformação, decorrente do ambiente Eclipse/Ecore e da linguagem de transformação ATL.

Concluímos que a técnica conseguiu detectar claramente problemas de entendimento da especificação formal da transformação, na fase pré-aplicação das regras, o que, algumas vezes, levou a uma construção errônea da tabela sintática.

A ameaça à validade desse experimento reside no poder de expressividade da LTL. Sabemos que outras lógicas temporais podem ser mais adequadas para certos tipos de propriedades específicas.

Finalmente, iniciamos o último estágio dessa avaliação de transformação PIM-para-PIM, onde vamos responder às questões relativas aos problemas semânticos encontrados pela técnica e sua classificação.

Em Maude, pudemos representar essa estrutura de comportamento dos modelos como um sistema de transições implícito nas configurações dos espaço de estados dos modelos. Um passo de transição entre dois conjuntos de estados $[s]$ e $[s'] \in T_{\Sigma/E}$ (elementos denotacionais de um sistema de reescrita) se e somente se existir um s_0 representativo $\in [s]$ e uma reescrita de um passo com a regra $R, s_0 \rightarrow 1s'_0$, tal que $s'_0 \in [s']$. Os dados que suportam essa interpretação são os várias espaços de estados já apresentados ao longo deste documento.

A verificação pela solução se deu através da verificação de propriedades de *segurança*, que são geralmente especificadas através de invariantes, e propriedades de vivacidade através de especificação de propriedades em lógica temporal. Vejamos a justificativa para os dois casos:

1. Dado um sistema Kripke e um estado inicial s_0, uma invariante é um predicado definindo um subconjunto de estados satisfazendo duas propriedades: contém s_0 e contém qualquer

estado alcançável a partir de s_0 através de um número finito de transições. Dessa forma, em Maude, como uma teoria de reescrita R = (Σ,E,ϕ,R), uma invariante I é definida por um conjunto decidível de estados, um estado inicial nesse conjunto e uma condição booleana definida equacionalmente I. O próprio manual da ferramenta Maude estabelece a notação: R,s_0 |= [] I para dizer que o sistema de transição associado com o conjunto de regras R e estado inicial s_0, satisfaz globalmente (operador LTL []) a invariante I.

2. Maude provê um módulo chamado MODEL-CHECKER capaz de prover algoritmos para busca utilizando os principais operadores da lógica LTL. Além do mais, provê um módulo chamado SATISFACTION, que permite a definição de proposições (chamadas por p) que devem ser válidas sobre o sistema em questão. Assim, as definições básicas de satisfação de proposição (p) e de caminhos (π) são dadas por: $T_{\Sigma/E}$,p |= ϕ e $T_{\Sigma/E}$,π |= ϕ respectivamente.

Código 6.1 em Maude. Predicados para o Controlador do Parque de Estacionamento

```
1:  mod PARKING-LOT-PREDS is
2:    protecting PARKING-LOT .
3:    including SATISFACTION .
4:    subsort Marking < State .
5:    ops ENABLED ARRIVE ENTER EXIT LEAVE : -> Prop .
6:    var Any : Marking .
7:    eq GateOpen Any |= ENABLED = true .
8:    eq EntFree Any |= ENABLED = true .
9:    eq FreePcs WaitingTicket Any |= ENABLED = true .
10:   eq CarInZone WaitingToPay Any |= ENABLED = true .
11:   eq GateOutOpen Any |= ENABLED = true .
12:   eq ExitFree Any |= ENABLED = true .
13:   eq Any |= ENABLED = false [owise] .
14:   eq EntFree Any |= ARRIVE = true .
15:   eq WaitnTkt Any |= ARRIVE = true .
16:   eq WaitnPay Any |= LEAVE = true .
17:   eq ExitFree Any |= LEAVE = true .
18:   eq GtOpen CarInZone Any |= ENTER = true .
19:   eq FreePcs GtOutOpn Any |= EXIT = true .
20:endm
```

Concluímos que Maude está apta a representar estruturas Kripke como modelos semânticos *que descrevem sistemas concorrentes. Além do mais, a ferramenta Maude model-checker provê muitos operadores de verificação inerentes. A facilidade de especificar propriedades em LTL tornou o uso dessa linguagem adequado à nossa solução.*

A ameaça à validade desse experimento reside em algum caso de ineficiência de implementação desses algoritmos de verificação por parte da ferramenta.

Para que pudéssemos trabalhar com especificações cada vez mais complexas, era necessário uma garantia de que a linguagem utilizada para especificação de propriedades pudesse ser composicional. Além do mais, o Maude precisa dar um suporte adequado ao nível de implementação e de usabilidade desses conceitos. Para que possamos fazer as devidas análises, vamos experimentar com o módulo de especificação de propriedades, apresentado no Código 6.1, gerado para o exemplo do controlador de estacionamento.

Esse módulo, declarado na linha 1, define os predicados do exemplo do parque controlador de estacionamento para todos os modelos envolvidos nas transformações. A seguir, temos inclusão dos módulos necessários (linhas 2 e 3) e a definição de marcação como estado da estrutura Kripke (linha 4). A seguir (linha 5), definimos os predicados que nos interessam de trabalhar ao nível dessa especificação. Para começar (linhas 7 a 13), temos a definição da propriedade ENABLED, que checa se pelo menos uma das condições para a rede estar viva é satisfeita. As propriedades ARRIVE (linhas 14 e 15) e LEAVE (linhas 16 e 17) checam a detecção de um carro por um desses controladores. Finalmente, as propriedades ENTER (linha 18) e EXIT (linha 19) identificam se um carro está na zona interna ou externa de um estacionamento.

A partir dessas proposições básicas, podemos definir novas proposições como combinação dessas:

1. `eq ARRIVE /\ ENTER`. Especifica que em um dado estado temos que um carro chegou e que está dentro do parque.

2. `eq ARRIVE -> ENTER`. Especifica que se um carro tiver chegado, implica em dizer que o carro já estará dentro do parque.

3. `eq LEAVE /\ EXIT`. Especifica que em um dado estado temos que um carro saiu e que está fora do parque.

4. `eq LEAVE -> EXIT`. Especifica que se um carro tiver saído, implica em dizer que o carro estará fora do parque.

5. `eq ([] ENABLED) -> <> (ENTER \/ EXIT)`. Especifica que se acontecer de todos os estados estarem habilitados, então em algum momento adiante teremos um movimento de entrada ou de saída de carros.

6. `eq [](ENTER -> <> EXIT)`. Especifica para todos os estados que, se um carro entrar então em algum momento no futuro, ele deverá sair.

Verificou-se que, devido às saídas intuitivas apresentadas pela ferramenta, pode-se combiná-las com outras ferramentas de interpretação visual de grafos, aumentando assim, o nível de usabilidade da ferramenta. Além da visualização completa do espaço de estados, outras características são apresentadas. Por exemplo, a solução permite desencadear uma prova, através de um caminho de estados, de que uma certa propriedade acontece. Retomando à busca pela propriedade de deadlock, a Figura 55 demonstra a existência de um caminho circular no qual a rede poderia ficar viva eternamente em caso de repetição desse ciclo. Como o espaço de estados é de uma ordem muito grande para esse modelo, esses caminhos de contra-exemplo gerados e apresentados ajudam a dar uma clareza maior sobre a veracidade das provas.

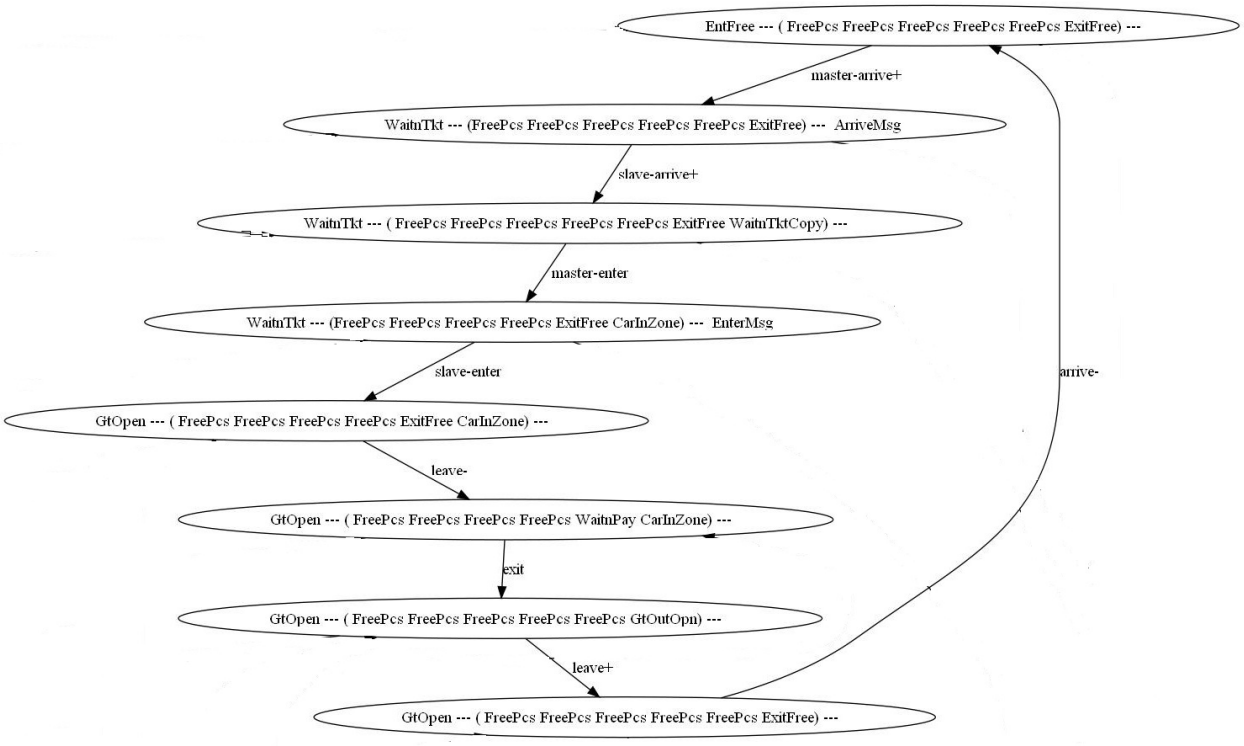

Figura 55: Caminho encontrado no espaço de estados para o modelo de saída da transformação

Concluímos que a solução garante uma avaliação em caso de não preservação de propriedades que vai além de apenas observar os resultados lógicos da verificação. Ela permite a observação dos contra-exemplos fornecidos, aumentando a precisão por verificação de equivalência.

A ameaça à validade desse experimento reside novamente na interpretação errônea dos resultados lógicos e contra-exemplos fornecidos.

Com um cenário de verificação bem definido, apresentamos aqui um resumo das características de problemas semânticos que a solução conseguiu detectar devido a problemas de implementação da transformação. Isto é relatado segundo interpretações dadas pelo cliente MDA sob o critério de preservação de semântica:

- *Número de transições mestre maior que um*. Embora esse problema já tenha sido tratado ao nível sintático, constatou-se que seu problema ia além desse nível por erros no entendimento da especificação. O problema é que um conjunto de sincronização pode conter um número maior que um de transições escravas e apenas uma transição mestra. Contudo, em alguns casos nos modelos, esse conjunto de transições era particionado, necessitando de um número transições mestras maior que um para que satisfizesse cada subconjunto de transições escravas. Este problema foi muito freqüente no modelo do parque de estacionamento de dois andares e foi identificado através da verificação de seqüência de eventos.

- *Erro de cópias entre nós, criando ou excluindo ligações entre elementos disjuntos*. Este problema foi capturado com a busca de vivacidade entre os modelos, haja visto que apresentaram comportamentos diferentes sob esse critério. Um exemplo disso foi no parque de estacionamento de um andar, no qual o arco da transição `Arrive+` para `WaitingTicket` não foi gerado.

- *Erro de mapeamento entre marcações. O espaço de estados dos modelos não representavam uma estrutura adequada*. Isso foi detectado através da busca de deadlock para os dois exemplos do parque de estacionamento.

- *Cruzamento de canais*. Canais de comunicação tinham uma semântica de execução da especificada pelas IOPTs. Em especial, houveram casos de um assincronismo com estados intermediários para um cenário em que a comunicação síncrona era esperada. Isso foi detectado pela busca de seqüência de eventos estabelecendo ordem entre transições mestre e escravas.

- *O conjunto LTS é diferente do Tm U Ts especificado para o mapeamento*. Esse tipo de problema foi identificado pela aplicação de alguns padrões de especificação LTL. Por exemplo, não se verificou um conjunto vazio de estados quando procurando pela propriedade de *ausência entre Q e R*, com Q envolvendo uma transição pertencente a Tm e R envolvendo uma transição pertencente a Ts.

- *Modelos assíncronos em deadlock*. Devido à necessidade de geração de eventos para modelos particionados, ocorreu muitas vezes de esses modelos não possuírem eventos cadastrados corretamente em suas transições mestre e escravas, de forma que, ocorria uma parada completa do sistema diferindo, a esse nível de comportamento, do modelo de entrada.

- *Erro de reposicionamento do conjunto de corte*. Durante a execução de verificação de propriedades segundo padrões, descobriu-se em vários casos uma má formação dos elementos pertencentes ao conjunto de corte no modelo destino. Isso ocorreu, por exemplo, ao pesquisar-se pelo padrão universalidade, em que todos os elementos do conjunto de corte teriam que aparecer universalmente disponíveis no modelo de saída.

Para respondermos à questão corrente, ainda continuamos recorrendo aos dados apresentados anteriormente. Por exemplo, no último item de problema semântico apresentado (*erro de reposicionamento do conjunto de corte*), nos referimos às buscas utilizando o padrão de especificação LTL de universalidade, que requisitava a presença constante de propriedades que executavam as transições existentes no conjunto de corte. Essa afirmação só ficou clara, devido ao padrão universalidade generalizar todas as fórmulas presentes na sua definição que é complexa.

A comprovação dessa afirmação vem com os resultados que obtivemos com a análise de mutantes empregada ao cenário de avaliação. Selecionamos os seguintes operadores que obtiveram os seguintes resultados resumidos em seu escore de mutação (ms). Alguns dados que garantem essas evidências podem ser encontrados no site do projeto [MDA-VERITAS, 2011].

1. InitMarkDel. Esse operador apresentou ms = 1 em detecção de preservação de propriedades;

2. InitMarkIns. Esse operador apresentou ms = 0.95 em detecção de preservação de propriedades;

3. InitMarkExchange. Esse operador apresentou ms = 0.88 em detecção de preservação de propriedades;

4. InputArcInversion/OutputArcInversion. Esse operador apresentou ms = 0.72 em detecção de preservação de propriedades, produzindo resultados inadequados para nossa avaliação. Isto se deve devido ao fato de haver mudanças de características essenciais de comportamento nos modelos quando se inverte arcos. Tem-se que descartar praticamente tudo que havia sido inferido pelo modelo anterior;

5. InputArcDel. Esse operador apresentou ms = 1 em detecção de preservação de propriedades;

6. InputArcExchange. Esse operador apresentou ms = 0.95 em detecção de preservação de propriedades;

7. OutputArcExchange. Esse operador apresentou ms = 0.95 em detecção de preservação de propriedades;

Concluímos que foi possível classificar problemas semânticos encontrados de acordo com o tipo de propriedades que os modelos satisfizeram ou deixaram de satisfazer. Além do mais, o uso de padrões de especificação permitiu uma adequação da solução ao uso de clientes não familiarizados com as complexidades envolvidas nas especificações em lógica temporal. Agora, há uso desses padrões por parte de todos os atores envolvidos no emprego da solução. Finalmente, os testes de mutação dos modelos, apresentaram que mesmo em situações inesperadas, a solução consegue lidar com a verificação de preservação de semântica de forma satisfatória com um alto índice de captura de equivalência.

A ameaça à validade desse experimento reside novamente no uso incorreto desses padrões de verificação.

6.3.2 Resultados em Verificação de Modelos em uma Transformação PSM-para-PSM

Para apresentação dos dados referentes à avaliação de transformações PSM-para-PSM, nos limitamos mais à apresentação de dados e aos comentários que acrescentem informação às interpretações já realizadas por esse trabalho. A seção resume-se à sumarização mesclando uma visualização envolvendo resultados específicos ao abordar os objetivos e à introdução de gráficos que resumem séries de tabelas que necessitariam de uma maior discussão.

O exemplo que guia esta avaliação da técnica em transformações PSM-para-PSM é um controlador domótico, conforme sua apresentação que se deu no Capítulo 5.2. O sistema poderia dividir-se em vários módulos, representando um aspecto específico de controle para cada caso, respectivamente.

A Tabela 22, de execução da transformação, inicia a apresentação das respostas às questões temporais enunciadas no *Objetivo 1*. A transformação, nesse caso, partiu de M_{in} possuindo a representação *IOPT com Semântica Maximal Step e Eventos* e direcionou-se para M_{out} possuindo a representação *Globalmente Assíncrono e Localmente Síncrono (GALS)*. Resultados de aplicação

com outras variações de semântica para PSMs não são relevantes devido ao fato de todos os valores serem similares. Neste caso, o que mais influencia o desempenho é a característica estrutural da rede, que é uma variável igual em todos os contextos.

Execuções	TT1	TTS1	TT2	TTS2	TT3	TTS3	TT4	TTS4	TT5	TTS5	TT6	TTS6
1	1.2s	0.3s	1.2s	0.3s	1.5s	0.5s	1.6s	0.5s	1.7s	0.6s	1.9s	0.6s
2	1.1s	0.3s	1.2s	0.3s	1.5s	0.5s	1.6s	0.5s	1.7s	0.5s	1.9s	0.6s
3	1.1s	0.4s	1.2s	0.3s	1.5s	0.5s	1.6s	0.5s	1.7s	0.5s	1.9s	0.6s
4	1.1s	0.4s	1.2s	0.3s	1.5s	0.5s	1.6s	0.5s	1.7s	0.6s	1.9s	0.6s
5	1.2s	0.3s	1.2s	0.3s	1.5s	0.5s	1.6s	0.5s	1.7s	0.5s	1.9s	0.6s
Média	1.1s	0.3s	1.2s	0.3s	1.5s	0.5s	1.6s	0.5s	1.7s	0.5s	1.9s	0.6s

Tabela 22: Execução da transformação Splitting para modelos do Controlador Domótico

As médias totais de aplicação da transformação (TT) para as várias plataformas difrentes foram de 1.195, 1.227, 1.561, 1.650, 1.766 e 1.976 segundos respectivamente. As aplicações da extração da tabela sintática (TTS) ficaram em 0.395, 0.364, 0.524, 0.578, 0.591 e 0.616 segundos respectivamente. Isto representa aproximadamente 33% do tempo total de aplicação. Novamente, *a extração da tabela sintática não acrescentou tempo significativo à transformação*.

Para o item Desempenho, tomamos as métricas correspondentes, através das *equações semânticas*. Os tempos estão apresentados na Tabela 23 no mesmo formato da seção anterior.

Execuções	Transf para o modelo semântico$_{in}$	Geração de código$_{in}$	TEMS$_{in}$	TEMS$_{out}$
1	1.614 s	0.419 s	1.844 s	2.002 s
2	1.523 s	0.422 s	1.802 s	1.904 s
3	1.512 s	0.434 s	1.793 s	1.993 s
4	1.481 s	0.411 s	1.821 s	1.859 s
5	1.509 s	0.410 s	1.817 s	1.926 s
Média	1.527 s	0.419 s	1.815 s	1.936 s

Tabela 23: Extração das equações semânticas para modelos do Controlador Domótico

Com esses resultados, o processo de extração das *equações semânticas* permanece viabilizado. A ordem do tempo de extração permanece inferior à complexidade do modelo. Além do mais, no caso dos PSMs, muitos componentes do código a ser utilizado pelo *modelo semântico* é existente, sendo apenas acoplado ao código gerado. A geração em si para PSMs tem a mesma complexidade da apresentada para PIMs e isso vale para todos os possíveis *modelos semânticos* apresentados para PSMs.

A verificação de propriedades, por outro lado, apresenta resultados com variações bem maiores de acordo com a semântica de execução escolhida. Para efeitos de simplificação das informações apresentadas nos gráficos e tabelas a seguir, utilizaremos as seguintes indexações para o tipo de semântica de execução (SE) correspondentes para os modelos semânticos:

- SE.1: IOPT com Semântica Maximal Step e Eventos;
- SE.2: IOPT com Semântica Maximal Step e Eventos Exclusivos;
- SE.3: IOPT Síncrono com Clock Global;
- SE.4: Mensagens Síncronas e Módulos Assíncronos;
- SE.5: Semântica Interleaving e Mensagens Assíncronas;
- SE.6: Globalmente Assíncrono e Localmente Síncrono.

A Tabela 24 apresenta os resultados das execuções para todos os tipos de semântica de execução mencionados.

Execuções	SE.1	SE.2	SE.3	SE.4	SE.5	SE.6
1	0.53 s	0.56 s	0.41 s	1.07 s	0.95 s	1.38 s
2	0.55 s	0.58 s	0.39 s	1.05 s	0.92 s	1.40 s
3	0.52 s	0.57 s	0.41 s	1.10 s	0.91 s	1.41 s
4	0.52 s	0.58 s	0.42 s	1.10 s	0.95 s	1.44 s
5	0.52 s	0.57 s	0.42 s	1.11 s	0.94 s	1.44 s
Média	0.53 s	0.57 s	0.41 s	1.09 s	0.94 s	1.42 s

Tabela 24: Verificação de deadlock para todas as semânticas de execução de PSMs

Quando alteramos o número de marcas no lugar `waitingToEnable` percebemos variações mais significativas no tempo de verificação da propriedade. Isto significa evoluir o tamanho do modelo, aumentando o número de habilitações concorrentes existentes no modelo. O gráfico na Figura 56 apresenta a evolução temporal em termos do número de tokens para todos os modelos semânticos envolvidos. Percebemos que há variações significativas dependendo do *modelo semântico* adotado. Isto se deve ao fato da busca por deadlock ter de percorrer todo o espaço de estados do sistema. Vê-se a alta ordem de crescimento da componente tempo para a plataforma GALS, devido à complexidade do comportamento desse modelo. Em outros casos, como a semântica com

maximal step e eventos ou IOPT síncrono com clock global, o crescimento da componente tempo é bem mais moderado.

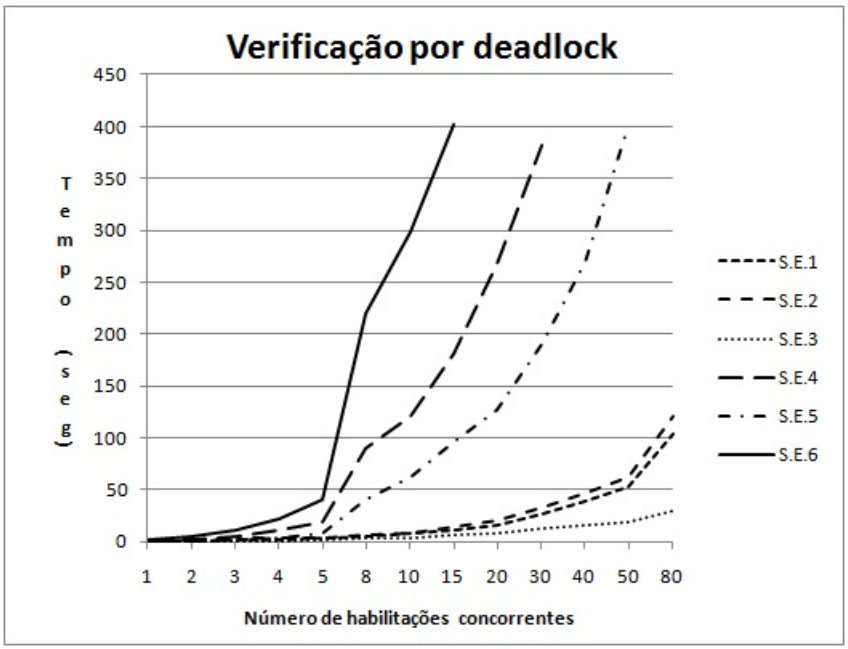

Figura 56: Resultados temporais para diferentes semânticas de execução

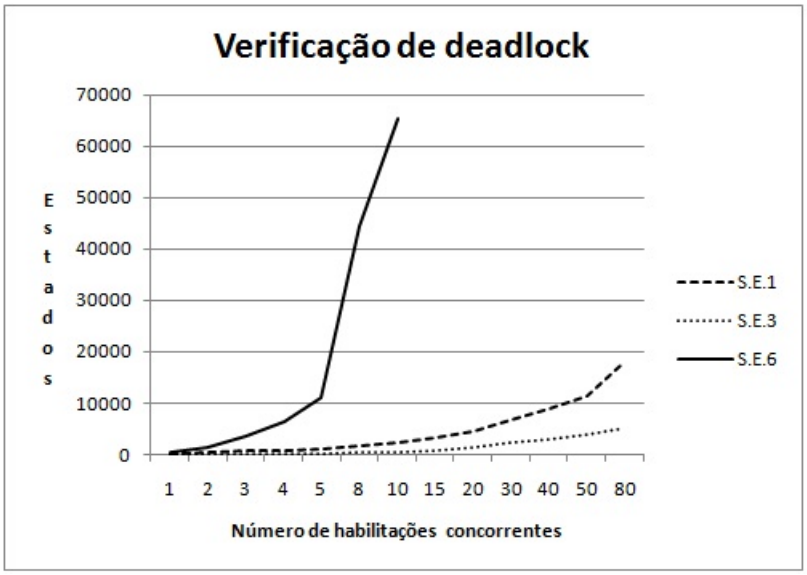

Figura 57: Resultados espaciais para três semânticas de execução

A evolução do número de estados das soluções também apresentam um crescimento considerável. A Figura 57 os apresenta com a simplificação para os três casos de semânticas de execução correntemente analisados, iniciando nossas observações sobre o *Objetivo 2*. Percebe-se que as

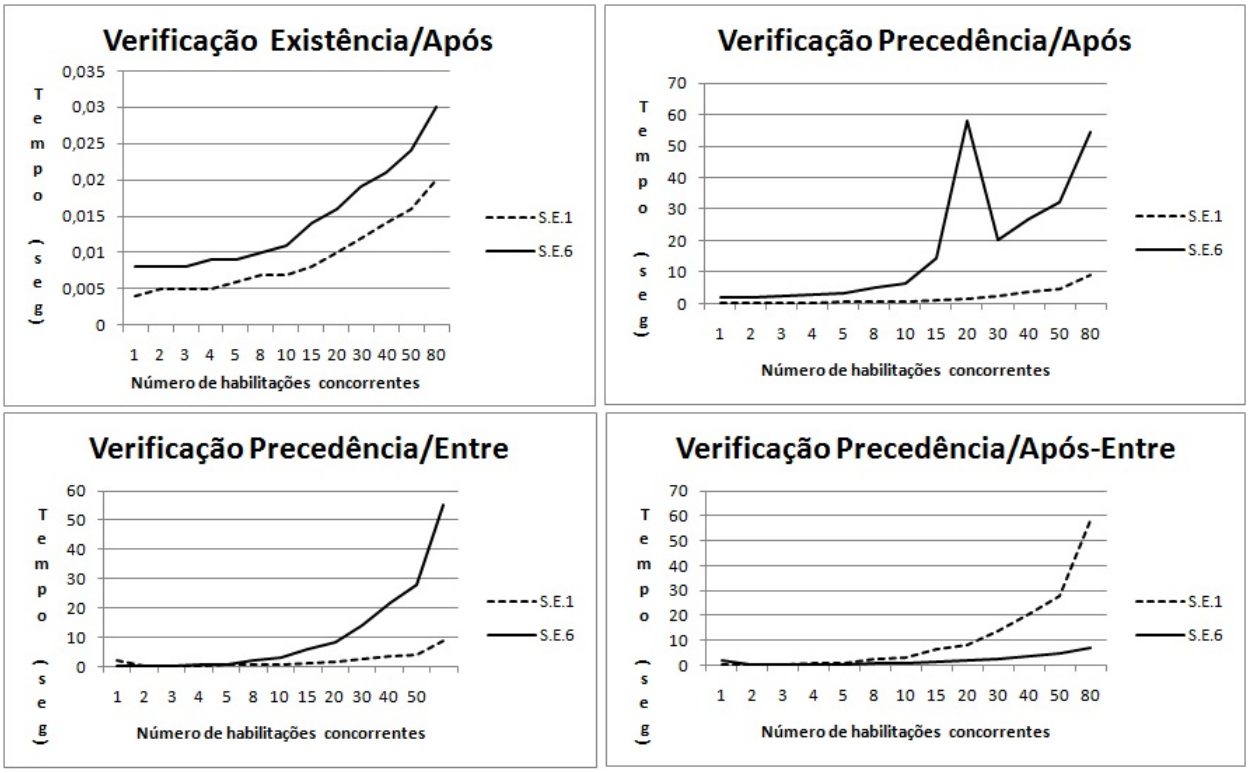

Figura 58: Resultados temporais para duas semânticas de execução sob várias propriedades

curvas de evolução no tempo são quase que idênticas.

Na verificação de propriedades, temos que essa tendência não se repete. O comportamento será muito específico de acordo com o tipo de fórmula LTL empregada [Katoen, 1999], como pôde-se observar nas avaliações do *Objetivo 3* e do *Objetivo 4* apresentados anteriormente. Para exemplificar isto, a Figura 58 apresenta a evolução no tempo de propriedades distintas com comportamentos bem diferenciados umas das outras. Lembrando que estas propriedades e seus padrões foram responsáveis diretamente pela categorização dos problemas sintáticos e semânticos que pudemos encontrar quando aplicando a solução. O gráfico da parte superior esquerda apresenta uma curva de crescimento aproximada, com alguma superioridade para a semântica GALS (SE.6). Já o gráfico da parte superior direita apresenta fortes distorções em um comportamento (SE.6), enquanto que a curva de crescimento apresentado pelo outro comportamento (SE.1) permanece previsível. Da mesma forma, não se verifica nenhuma tendência nos dois gráficos da parte inferior. Ao lado direito, temos um crescimento acentuado da SE.6, enquanto que ao lado esquerdo, temos um crescimento acentuado por parte da SE.1. Isto vai depender diretamente do resultado da propriedade semântica verificada e se esse resultado necessitou da construção de contra-exemplos.

Ainda sob o propósito da verificação de propriedades, e de acordo com o *Objetivo 2*, podemos

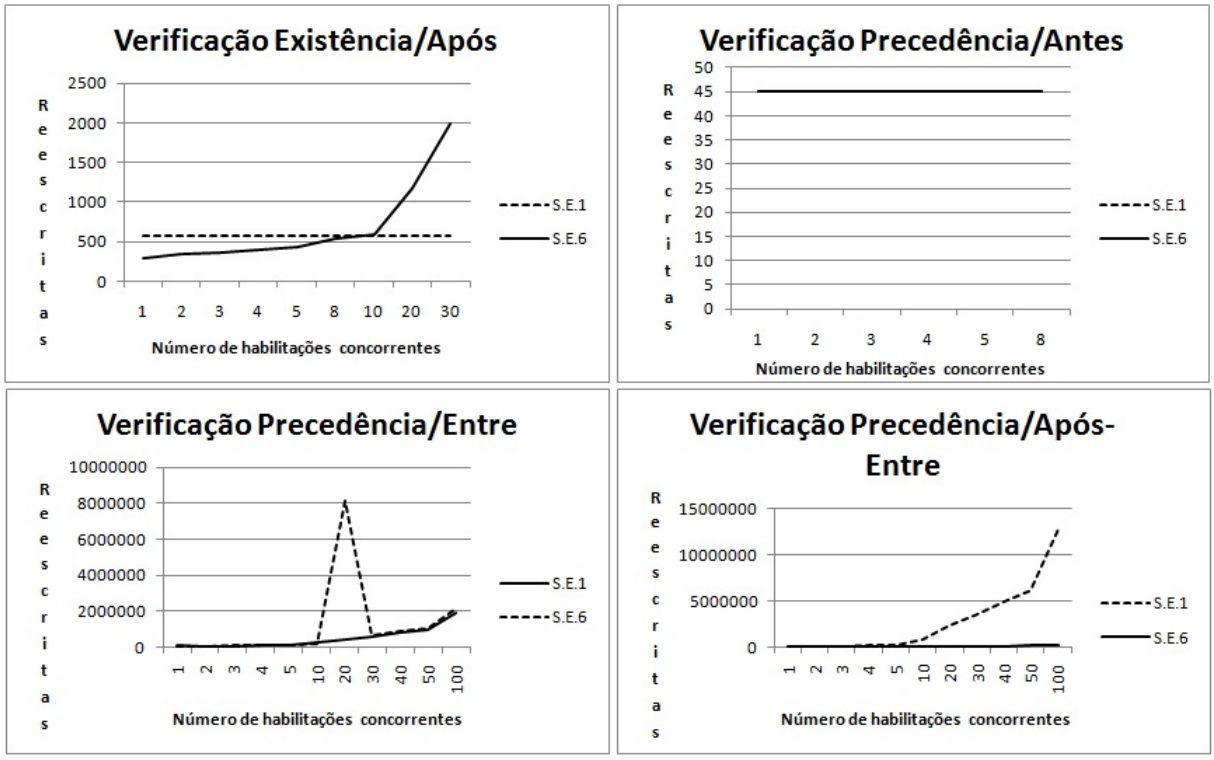

Figura 59: Resultados espaciais como reescritas para duas semânticas de execução sob várias propriedades

avaliar a capacidade de escalabilidade de acordo com o número de reescritas na verificação das propriedades. Os quatro gráficos apresentados na Figura 59, equivalentes às propriedades verificadas na Figura 58, comprovam que também no critério espaço, cada caso dependerá da propriedade a ser avaliada. Na parte superior da Figura 59, temos dois gráficos onde o número de reescritas de SE.1 permanece estável mesmo com o aumento do tamanho dos modelos. Porém, para o caso de SE.6 poderemos ter um crescimento com diferentes taxas, como é o caso do gráfico da esquerda, ou também uma estabilidade, como é o caso do gráfico da direita. Na parte inferior da Figura 59, também observamos comportamentos diferenciados. No gráfico da esquerda temos subitamente, em uma determinada configuração do modelo, um ponto diferenciado para o comportamento de SE.6, enquanto que no gráfico da direita, em um determinado momento na evolução de tamanho dos modelos, o número de reescritas de SE.6 abandona a estabilidade que era seguida por SE.1, apresentando elevadas taxas de crescimento no número de reescritas.

Finalmente, os resultados da análise de mutantes para este cenário de avaliação de transformação PSM-para-PSM foi similar ao caso anterior, de transformação PIM-para-PIM, sob todos os operadores. Isto significa dizer que para este caso a solução também conseguiu lidar com a

verificação de preservação de semântica de forma satisfatória com um alto índice de captura de equivalência.

6.4 Recapitulação e Considerações

Este capítulo apresentou avaliações experimentais para o uso da instanciação da MDA-Veritas para transformações envolvendo modelos de sistemas concorrentes. A avaliação abordou modelos PIMs e modelos PSMs em várias plataformas e semânticas de execução. Os resultados comprovaram a viabilidade das técnicas e ferramentas escolhidas para o contexto abordado.

Detalhando essa comprovação sobre a viabilidade, descobrimos e detalhamos os seguintes fatos sobre nossa solução:

1. A técnica de extração de uma tabela sintática não acrescenta tempo significativo à transformação sob questão original;
2. O crescimento de complexidade do modelo pouco influencia no tempo de extração dos modelos semânticos através das equações semântica;
3. A complexidade do modelo influenciará no custo de aplicação das regras de computação e consequentemente no tempo de verificação;
4. A ferramenta de verificação adotada é eficiente se comparada à ferramenta de referência desta área;
5. O tamanho dos modelos analisados foi suficiente para representação de aplicações do cliente MDA;
6. O problema da verificação esteve dentro da complexidade assintótica esperada;
7. Classificamos os tipos de propriedades sob verificação, aumentando o nível de abstração;
8. Capturamos erros no entendimento de especificações;
9. Conseguimos representar comportamento de modelos em ordens elevadas;
10. Capturamos erros esperados quando submetida à exaustão de variações introduzidas por mutantes;

11. Provemos contra-exemplos que facilitaram o entendimento do usuário sobre o tipo de erro ocorrido;

12. Comprovamos a eficiência em captura de erros da técnica através de cenários com muitas variações, fazendo uso da análise de mutantes.

A avaliação PIM-para-PSM foi omitida devido ao fato do método parecer redundante, pois repetiria o processo de extração de PIMs e de PSMs, o que já foi devidamente explorado aqui.

Maiores detalhes sobre a metodologia que envolveu esse estudo experimental podem ser encontrados em [Costa et al., 2010]. Finalmente, para detalhes complementares envolvendo métricas e avaliações específicas para a transformação Splitting e modelos em redes de Petri como sugere a instanciação, recomendamos que leitor o consulte ao documento *Avaliação Experimental Estendida*, disponível em [MDA-VERITAS, 2011].

7 Trabalhos Relacionados

Para a construção deste trabalho, estudamos diversas idéias levantadas por alguns trabalhos relacionados. Cada trabalho em particular apresenta um aspecto e uma contribuição específica para um determinado ponto da nossa idéia. Antes de introduzir uma lista de tópicos no qual vários trabalhos apresentam uma parcela de contribuição, devemos primeiramente nos concentrar no trabalho que nos motivou na busca pela formalização da idéia da concepção da *arquitetura MDA estendida*. Este trabalho chama-se *dynamic metamodeling* (*metamodelagem dinâmica*) [Hausmann, 2005]. Ele foi proposto originalmente por [Engels et al., 2000] e foca na especificação de relações entre metamodelos (sintáticos e semânticos) como uma forma de prover semântica para os modelos. Esta semântica pode ser dada por diferentes formalismos ou domínios semânticos.

Como um exemplo de instanciação do trabalho mencionado, os autores apresentam uma visão da técnica para uso da linguagem OCL. Eles descrevem claramente o conceito da semântica denotacional, com um mapeamento entre o mundo sintático e semântico. O problema é que esse mapeamento é feito através da associação simples de pares. No sentido oposto, nossa abordagem instancia tais semânticas unificando-as para apenas um formalismo em um mesmo paradigma, ou seja, os modelos semânticos de entrada e de saída devem ser expressos pelo mesmo domínio semântico com o objetivo de executar operações semânticas no intuito de compará-los. Desta forma, nosso trabalho diferencia-se por uma unificação de domínios semânticos. Observe que isso não implicou em dizer que as linguagens necessitariam ser as mesmas, mas o seu formalismo semântico precisa ser unificado. Além do mais, propomos o uso de um *Verificador Formal* que permita executar operações semânticas em ambos modelos e metamodelos. Isto é alcançado através de técnicas de reescrita e técnicas formais de verificação, provendo mais resultados que as transformações de grafos propostas em [Hausmann, 2005]. Então, a aplicação de nossa abordagem não está restrita a dar semântica para modelos de entrada, mas também para verificar equivalência semântica entre modelos de entrada e de saída.

Logo depois, o trabalho desenvolvido em [Bézivin et al., 2006] reusou a idéia da divisão entre os mundos sintáticos e semânticos em MDA para apontar uma perspectiva das transformações entre modelos como modelos de transformações. Desta forma, pode-se argumentar sobre benefícios dessa abordagem no campo da verificação. Apesar de não relatar como essa verificação seria realizada, os autores discutem sobre as propriedades semânticas que uma transformação pode ter ao acessar os modelos de entrada e de saída da transformação. As propriedades são puramente

baseadas em restrições definidas e são de responsabilidade do desenvolvedor da transformação. Em nosso caso, vamos adiante sobre estas propriedades dos modelos porque nos interessamos pelo comportamento dinâmico dos mesmos. Obviamente, esta atividade é mais delicada e não pôde ser definida pelo desenvolvedor da transformação que chamamos de *Cliente MDA*. Esta responsabilidade fica com o *Provedor da Solução* que deverá ter um conhecimento aprofundado sobre o paradigma dos modelos envolvidos na transformação e quais propriedades dinâmicas interessam, sendo geradas ou especificadas manualmente.

Atualmente, o trabalho [Hülsbusch et al., 2010] redefiniu o problema que atacamos de acordo com os termos levantados por este trabalho. Contudo, a solução empregada pelos autores novamente recorre ao método de definir uma noção de equivalência para transformações. Em nosso trabalho, mais precisamente na formalização do módulo *Verificador Formal*, nos concentramos em uma justificativa para a nossa noção de equivalência tomando este trabalho como elemento comparativo devido ao fato de abordar uma transformação já construída.

Ao compararmos o nosso interesse com o de [Hülsbusch et al., 2010], a abordagem escolhida por este trabalho implicou em uma série de benefícios e desvantagens, que tivemos de lidar ao longo dessa tese:

- Completude: perdemos sob esse aspecto devido ao fato de desejarmos nos concentrar em instâncias específicas de modelos para uma dada transformação. Assim, não garantimos que, para todos os casos de modelos de entrada da transformação, sua preservação de semântica ocorra.

- Aplicação: obtivemos uma aplicação imediata em projetos acadêmicos e industriais que puderam oferecer um fluxo de desenvolvimento baseado em MDA. A noção de equivalência permite fazer uso da técnica sobre os modelos mesmo não tendo disponível a transformação de imediato.

- Flexibilidade: a equivalência a ser verificada pode ser especificada ao nível de preservação de propriedades entre os modelos envolvidos na transformação. Desta forma, a noção de preservação de semântica pode ser ajustada de acordo com os interesses do cliente da solução, seja ao nível de estrutura, significado, comportamento ou inúmeras outras formas.

- Caracterização semântica: Uma abordagem que busque a completude, naturalmente tende a não dar muita importância às características particulares de uma determinada instância,

pelo fato de ter que se caracterizar o conjunto, ou domínio, como um todo. Com isso, casos particulares da aplicação das regras da semântica operacional podem não ser levados em consideração, dado que as transformações e os metamodelos são artefatos mais importantes a serem considerados. Isto faria com que ainda se concentrassem em questões sintáticas ou de semântica estática. Neste caso, a abordagem seria classificada como de semântica denotacional pura. Ao adotar-se uma semântica translacional para lógica de reescrita, por exemplo, reusa-se todos os seus componentes sob vários pontos de vista, tais como algébrico, operacional e outros mais, concentrando-se nas peculiaridades específicas apenas dos modelos envolvidos nas transformações.

Ainda existem algumas outras linhas de pesquisa de fundamental importância e que também devem ser mencionadas por este capítulo. De uma maneira geral, seus focos classificam-se pelas seguintes abordagens:

- Verificação formal de transformações de modelos.
- Verificação formal de modelos PIMs.
- Verificação formal de modelos PSMs.

Cada uma dessas abordagens merece uma sub-seção especial devido ao número de trabalhos encontrados e por diferenças complexas existentes entre eles que merecem uma explicação mais detalhada.

7.1 Verificação Formal de Transformações de Modelos

Como prova das diferenças de rumo que a pesquisa originadora da idéia de concepção desse trabalho ([Engels et al., 2000]) tomou, consideremos [Hülsbusch et al., 2010] como primeiro trabalho relacionado nessa categoria por representar o resultado mais recente desse grupo de pesquisa. O trabalho lida diretamente com o problema da falta de preservação semântica das transformações, e apresenta uma técnica bastante formal que, embora aplicada em cenários bem fora da realidade, são capazes de garantir uma preservação completa e é aplicada a transformações envolvendo modelos em linguagens diferentes. Este trabalho apresenta um exemplo que prova a bissimilaridade

entre esses modelos. O método para inferir isso é não-trivial e ainda não foi implementado. Diversos grafos possíveis de interpretação de comportamento são extraídos e essa relação entre eles é fraca ou parcial.

Esse tipo de abordagem baseada em transformação de grafos é bastante clara e formal, mas de difícil aplicação. O principal problema é a falta de ligação com o problema existente em MDA, já que o título do trabalho sugeriria isso. Dessa forma, problemas de escalabilidade, mecanização, confiança em níveis altos de criatividade e falta de suporte ferramental ficam evidentes. Além do mais, não há uma referência clara aos tipos de modelos de MDA que essa técnica contempla, tais como CIMs, PIMs e PSMs. Porém, o recente trabalho [Cabot et al., 2010] consegue utilizar bem essa abordagem de grafos ao derivar invariantes OCL a partir das regras de transformações declarativas. Estes resultados são empregados em verificação e análise, porém não tratam dos aspectos dinâmicos dos modelos envolvidos nas transformações. Ainda sobre restrições, [Petter et al., 2009] presta atenção à questão de caracterizar transformações através de restrições, integrando-as às relações QVT. Novamente, enfatizamos que a diferença deste tipo de trabalho com a nossa técnica consiste no interesse essencial que temos pelo comportamento dos modelos. Abordar a semântica dinâmica dos modelos é um diferencial em nosso trabalho, pois tende a ser muito pouco explorado por outros trabalhos na área.

[Baresi et al., 2006] afirma que transformações, se dadas de uma forma baseada em regras, como acontece em MDA, podem ser descritas como transformações de grafos. Eles tentam garantir preservação de semântica usando a ferramenta AGG no contexto de processos de negócio executáveis. Em Ehrig et al. [Ehrig et al., 2007], a preservação da informação é discutida com o requisito de bi-direcionalidade. Transformações de grafos são aplicadas para transformar diagramas de classes em modelos de bancos de dados relacionais. Por exemplo, o trabalho apresenta a derivação da regra de tripla de grafo para este caso. Na regra de avanço, uma nova tabela é criada para uma classe existente. A situação precisa também acontecer como vice-versa, onde na regra de retrocesso, uma tabela já existe e a classe correspondente deve ser criada. A principal diferença dessas abordagens para a nossa é a falta de análise estática e a definição de domínios semânticos através de metamodelos. Eles se concentram nas transformações apenas como regras operacionais, e não apresentam definições denotacionais dos modelos envolvidos nas transformações. Neste caso, o principal diferencial da nossa técnica é a abordagem semântica, pois nestes casos, apesar de envolverem conceitos dinâmicos, o processamento de verificação está restrita puramente a conceitos

sintáticos das transformações.

[Narayanan and Karsai, 2008] vai direto ao tópico apresentado, investigando se uma transformação de grafo preserva o comportamento para uma dada instância de entrada. Como um primeiro passo, tenta-se garantir essa conformidade com relação a uma determinada propriedade, como por exemplo, alcançabilidade. Isto é feito verificando se existe uma relação de equivalência entre os dois grafos. Essa checagem se dá através de bissimulação, provando que o modelo de entrada se comporta exatamente como o modelo de saída. Nosso trabalho, apesar de não utilizarmos bissimulação estritamente, representa uma evolução dessa técnica nos seguintes aspectos:

- *Vamos além de transformações de grafos*. Em nosso trabalho, propomos abordar transformações entre modelos analisando os modelos. Abordamos transformações envolvendo linguagens no paradigma concorrente e de acordo com o metamodelo da linguagem concorrente. O formato das transformações não nos interessa, bastando apenas que sigam o padrão QVT para extrairmos a *tabela de equivalência*.

- *Nos inserimos completamente em MDA*. Abordarmos QVT e as linguagens de transformação de MDA co-relatas. Também reusamos o padrão de definição de metamodelos MOF, a linguagem de restrições OCL e tantas outras técnicas e ferramentas. Assim, estamos progredindo na melhora de uma metodologia completa. Com relação ao trabalho mencionado, não foi encontrado relato de onde ele esteja sendo aplicado de fato.

- *Formalizamos modelos e metamodelos semânticos*. Esse é um outro diferencial da nossa abordagem. No trabalho relacionado não há uma extração do significado denotacional do modelo, havendo apenas simulações. Em nosso caso, vamos desde a concepção das estruturas que representarão o significado do modelo, propondo metamodelos semânticos de acordo com alguma técnica formal e a correta instanciação desses modelos.

- *Verificamos diversas propriedades*. No trabalho mencionado, apenas uma ou algumas propriedades específicas são sugeridas como prova de equivalência entre os sistemas para o caso da alcançabilidade. Abordamos técnicas mais sofisticadas, como verificação de semântica dinâmica dos modelos, proposta de uso de verificação de modelos, ou outras técnicas apropriadas, além da comparação de verificações no intuito de obter uma prova completa sob diversos ângulos da corretude da transformação.

Acrescentando valor aos resultados do trabalho anterior, [Narayanan and Karsai, 2006] propõe também o uso da técnica de *ancoragem semântica* para especificar a semântica dinâmica das linguagens. Agora, a verificação de comportamento se dá combinando bissimulação e ancoragem semântica, eliminando algumas das ausências apresentadas no tópico anterior. Os autores aplicam essa combinação de técnicas a duas variantes de *statecharts* da UML [OMG, 2011b], especificam suas semânticas operacionais e verificam para uma execução particular da transformação. De acordo com o seu framework para verificar preservação do comportamento, diversos conceitos importantes presentes em nosso trabalho podem ser observados, como bissimulação fraca e um framework semântico em comum. A diferença e desvantagem ainda consiste na relação formal existente de transformação de grafos com os artefatos de MDA. Essa ligação ainda não é provida de uma maneira clara, apesar de este trabalho apresentar um conteúdo formal bem mais sólido em comparação ao nosso.

7.2 Verificação Formal de modelos PIMs

Alguns trabalhos combinam técnicas formais de verificação na validação de modelos independentes de plataforma. O trabalho [Ray and Sumners, 2007] discute uma abordagem para permitir que duas técnicas diferentes, verificação de modelos e prova de teoremas, complementem uma à outra. Este trabalho permite automação em provas de invariantes, enquanto preservando a expressividade e controle provida por provadores de teoremas. Por exemplo, no caso de análise da propriedade de alcançabilidade temos que qualquer verificador de modelos pode resolver esse problema facilmente, contudo, o uso de provadores de teorema permite explorar abstrações no comportamento dos modelos de maneira bem mais eficiente, obtendo ganhos.

Como uma abordagem mais específica, [Mokhati et al., 2007] apresenta um framework que suporta verificação formal de diagramas UML (Unified Modeling Language) ao usar as capacidades orientadas a objetos e concorrentes do sistema Maude. O processo se resume a sistematicamente derivar especificações formais Maude a partir de análises dos diagramas de classes, de estados e de comunicação da UML. Ao final, tem-se a geração de diversos módulos do Maude. Apesar desse trabalho ter uma idéia consistente e similar à nossa abordagem, vale ressaltar que o nosso trabalho foca em paradigmas de modelos diferentes e analisa similaridades entre comportamentos de modelos envolvidos em transformações. Essa abordagem, que ressaltamos ser bastante valiosa, se resume à análise de um modelo individualmente.

No domínio de análise de modelos definidos por linguagens de programação, [Neuhausser and Noll, 2007] usa o framework de lógica de reescrita através do Maude para a verificação formal de modelos descritos na linguagem concorrente e funcional ERLANG. Os autores verificam propriedades implementando a semântica operacional formalizada desta linguagem. Esse trabalho possui resultados fundamentais que poderiam ser reusados no contexto da *MDA-Veritas* com instanciações para equacionais sobre o lambda-calculus.

Com relação à verificação de redes de Petri usando Maude, [Farwer and Leuschel, 2004] investiga Object Petri nets (OPNs), uma classe de redes de Petri que provê um método natural e modular para modelar muitos sistemas do mundo real. São realizados experimentos segundo uma abordagem de representação das redes no Maude e em Prolog. Diversos comparativos entre essas abordagens são realizados. Desta forma, esse trabalho seria complementar ao nosso, com uma nova técnica de verificação sobre redes de Petri no framework utilizado que pode ser incorporado ao Verificador Formal da MDA-Veritas.

[Stehr et al., 2001] propõe lógica de reescrita como um framework unificador para modelos em redes de Petri. Esse trabalho também apresenta um mapeamento das redes em especificações dadas em lógica de reescrita como assumido aqui. Por exemplo, a Figura 60 descreve um código Maude retirado desse trabalho. Nele, temos a definição de marcações e sua composição, além de lugares serem representados por operadores e transições por regras, como é similar ao nosso trabalho. A principal diferença desse para o nosso trabalho, neste caso, é que propomos este mapeamento como uma transformação MDA, observando regras sintáticas propostas no metamodelo *PNML*. No nível de PSM, tivemos que fazer representações bem mais complexas, inclusive recorrendo ao META-LEVEL da solução Maude. Neste sentido, nossa abordagem é bem mais pragmática, voltada a um domínio específico e sendo, por outro lado, menos geral.

A comparação do verificador de modelos do Maude é feita com várias outras ferramentas. Em [Ogata and Futatsugi, 2007], ele é comparado com as ferramentas SAL, que é um sistema para analisar sistemas de transições com ferramentas diferentes. Este trabalho detalha claramente as principais vantagens providas pelo verificador de modelos Maude. Ele reforça nossa escolha por esta solução de verificação de modelos.

Com relação ao paradigma concorrente, a linguagem de metamodelagem é formalizada nos resultados apresentados em [Meseguer and Montanari, 1988] e já detalhados anteriormente. A metodologia para sua implementação se dá em [Bruni et al., 2001], onde ao explicarem que a

```
sort Marking .

op empty : -> Marking .
op __ : Marking Marking -> Marking [assoc comm id: empty] .

ops BANK CREDIT-1 CREDIT-2 CLAIM-1 CLAIM-2 : -> Marking .

rl [GRANT-1]  : BANK CLAIM-1 => CREDIT-1 .

rl [RETURN-1] : CREDIT-1 CREDIT-1 CREDIT-1 =>
                BANK BANK BANK CLAIM-1 CLAIM-1 CLAIM-1 .

rl [GRANT-2]  : BANK CLAIM-2 => CREDIT-2 .

rl [RETURN-2] : CREDIT-2 CREDIT-2 =>
                BANK BANK CLAIM-2 CLAIM-2 .
```

Figura 60: Representação de uma rede de Petri em Maude

semântica algébrica de redes de Petri sob a filosofia de tokens coletivos pode ser explicada também em termos de tokens individuais, terminam por dar um tratamento categórico às redes e apresentam idéias de como isso também pode ser implementado no Maude. Aproveitando-se deste fato, reusamos estas idéias para definir e implementar nosso mecanismo de raciocínio semântico para as redes de Petri de Entrada e Saída. Este trabalho não visa preservação de semântica em transformações. A metodologia de construção de modelos semânticos é análoga a este trabalho e permite uma aplicação direta nos estudos de caso abordados.

Do ponto de vista do metamodelo sintático, [Ruscio et al., 2006] formaliza a linguagem KM3 usando *Máquinas de Estados Abstratas* em uma abordagem operacional. Além do mais, eles incluem um estudo de caso para dar à linguagem ATL uma semântica dinâmica. *Máquinas de Estados Abstratas* também são empregadas em [Chen et al., 2005] para a formalização das linguagens de modelagem específicas de domínio e de suas respectivas transformações no contexto de Computação de Modelo Integrado. Os autores, que já possuem referências inclusas na seção prévia, utilizam a técnica de *ancoragem semântica*, na qual é baseada na especificação transformacional de semântica, aplicada a uma plataforma de metamodelagem. Estes trabalhos empregam apenas a abordagem operacional. Outro interesse ocorre na especificação de mapeamentos entre a sintaxe abstrata e os domínios semânticos representando a abordagem denotacional. Além do mais, eles provêem semântica dinâmica como transformações de grafos. Por outro lado, nós trabalhamos com equações de reescrita no contexto da *teoria das categorias*. Finalmente, estas abordagens ainda sofrem da falta de verificação de se dois modelos estão de acordo conforme sua semântica.

Com relação à inserção de metamodelos para descrição de semântica formal, podemos relatar alguns trabalhos interessantes. Contudo, nenhum foca especificamente na álgebra semântica do formalismo denotacional. Kent et al. [Kent et al., 1999] redefine metamodelos para UML e OCL com o objetivo de que eles possam incorporar semântica formal. Neste caso, a semântica denotacional é realizada da seguinte maneira: (i) o metamodelo semântico para linguagens de modelagem (classes, papéis, modelos); (ii) o *metamodelo semântico* para a linguagem de instâncias (objetos, ligações); e (iii) o mapeamento entre essas duas linguagens como equações semânticas. Contudo, este trabalho apresenta diferenças em relação ao nosso pela falta de formalização das linguagens que descrevem os modelos envolvidos no domínio e no contra-domínio da transformação. Apenas o mapeamento não é suficiente para garantir conformidade entre modelos e dificulta considerarmos extensões para os aspectos dinâmicos do comportamento.

7.3 Verificação Formal de modelos PSMs

Os autores de [Chen et al., 2005] formalizam *Linguagens de Modelagem de Domínio Específico* e transformações no contexto de Computação Integrada a Modelo. Eles usam também a técnica chamada de *ancoragem semântica*, que se baseia em uma especificação transformacional de semântica, aplicada a uma plataforma de metamodelagem. Este trabalho emprega somente a abordagem operacional. Isto difere do nosso trabalho pelo fato de estarmos interessados também na especificação de mapeamentos entre sintaxe abstrata e domínios semânticos representando a abordagem denotacional. Além do mais, este trabalho provê semântica dinâmica como regras de transformações de grafos, diferenciando-se do nosso caso que é baseado em regras de reescrita. Outra grande diferença está em que nossa solução permite trabalhar, os nossos sistemas também permitem ser axiomáticos, pois estamos trabalhando com equações de reescrita tendo o uso de regras como opcional. Não necessariamente o conceito de estado deverá estar presente. Finalmente, essa abordagem ainda necessita de verificação de se dois programas resultantes de algum processo de transformação estão de acordo com suas semânticas.

O trabalho [Madl et al., 2006] aplica técnicas de verificação de modelos a sistemas embarcados de tempo real distribuídos usando transformações entre modelos para verificar propriedades chave de qualidade de serviço a sistemas baseados em componentes utilizado CORBA de tempo real. Neste caso, além das técnicas de verificação e de transformações, esse trabalho se assemelha ao nosso também pela introdução de um domínio semântico formal. Diversos conceitos com-

plexos são apresentados e utilizados, mas pode-se resumir que seu domínio semântico consiste de autômatos temporais. A principal diferença para o nosso trabalho é a falta de integração das plataformas e ferramentas, pois várias são utilizadas tais como DREAM (Distributed Real-Time Embedded Analysis Method), UPPAAL, KRONOs e outros mais. Em nosso caso, estamos orientados a resolver um problema específico de MDA. O domínio específico de sistemas embarcados foi uma instanciação do problema.

Existem dois trabalhos recentes que empregam Maude para formalizar modelos específicos de plataforma. [Boronat and Meseguer, 2009] adota Maude como uma metalinguagem, implementando alguns operadores genéricos para lidar com modelos EMF, como um suporte para o padrão relacional QVT para transformações. Isto foi o pilar para a concepção da solução MOMENT. Com as transformações declarativas direcionadas e a especificação algébrica, ferramentas formais podem ser usadas para raciocinar sobre algumas propriedades, tais como terminação e confluência [Boronat and Meseguer, 2008]. Estes aspectos também estão presentes em nosso trabalho, encorajando-nos a adotar as capacidades da lógica de reescrita. Em [Romero et al., 2007], os autores mostram como Maude provê uma forma acurada de especificar modelos específicos de plataforma e metamodelos com o suporte para raciocinar e executar as especificações. Maude permite a implementação de algumas operações nos modelos, tais como inferência de tipos e subtipagem. Esses trabalhos são relevantes para mostrar que o conjunto de ferramentas do Maude são adequados na comunicação com artefatos dos modelos quando tendo essa responsabilidade.

[Gheyi et al., 2007] aborda o problema da falta de critérios na proposta de refatoramentos. Isto dificulta prova de corretude com relação à semântica e permite a introdução de novos erros. Assim, os autores formalizam uma semântica estática para Alloy e a codificam no Prototype Verification System (PVS). Com isso, prova-se que os refatoramentos não introduzem erros de tipo. Em [Gheyi et al., 2005], os mesmos autores vão além da semântica estática e propõem uma semântica formal com uma linguagem base, removendo-se todos os auxílios sintáticos (*syntactic sugars*), e uma noção de equivalência. Eles afirmam que uma transformação preserva semântica se a semântica estática e a semântica dinâmica forem preservadas. Além de usar leis propostas em PVS, também fazem uso da ferramenta Alloy Analyzer para transformar modelos que contenham subtipos em uma versão otimizada para o escopo do trabalho. Assim, temos guias de como provar que refatoramentos estão corretos. Consideramos os modelos abordados como possuindo algumas características dessas plataformas, porém também podem enquadrar-se em uma visão independente

de plataforma.

Em [Li et al., 2005], os autores trazem as vantagens das transformações para o contexto dos programas funcionais, com um suporte específico para programadores Haskell. Eles construiram uma ferramenta chamada HaRe (Haskell Refactorer). A ferramenta se baseia na definição de um catálogo de refatoramentos definidos por eles mesmos. Eles vão além de experimentos com o lambda-calculus simples, como chegamos a realizar na concepção desse trabalho, porém as aplicações são apenas sintáticas, não havendo mecanismo algum de verificação de equivalência entre os códigos. Na verdade, devido ao alto grau de base formal que a programação funcional tem, consideramos que investigações sobre preservação de semântica ainda são muito pobres nesse domínio. Isto se deve ao fato de essas linguagens ainda não serem utilizadas de fato na indústria, gerando pouco interesse nesse tipo de investigação.

8 Considerações Finais

Este trabalho retomou um questionamento existente na comunidade MDA desde a concepção da sua idéia: *a preservação de semântica entre os modelos envolvido nas transformações*. Recorremos a modelos e estudos de caso tradicionais, além de uma parceria com um projeto (FORDESIGN) com contribuições científicas consolidadas e com forte interesse em aplicações industriais. Imediatamente após a caracterização dos resultados, pudemos aplicar o material existente no contexto industrial no ramo de automação inteligente de condomínios residenciais (domótica).

Propusemos uma extensão da arquitetura MDA instanciada para o domínio transformações envolvendo modelos de sistemas concorrentes descritos em redes de Petri, mas que, pode ser aplicada a outros contextos de transformações envolvendo modelos descritos em outras linguagens e paradigmas. Para que isto seja possível, convém ao leitor observar que, cada uma das atividades instanciação de módulos da MDA-Veritas são de livre escolha por parte do *Provedor da Solução*, requisitando apenas experiência e eficiência no uso de outras técnicas, padrões, linguagens e ferramentas que auxiliem na verificação como um todo. Sumarizando, para cada uma das atividades, temos as seguintes considerações:

1. *Prover regras de boa formação em um ambiente estático.* Esta atividade é inerente à construção de qualquer compilador. Então, o custo de se instanciar esta atividade não está atrelado à nossa técnica especificamente, pois assumimos que modelos que possuam propriedades interessantes de serem verificadas já deveriam possuir estas regras explicitamente como requisito mínimo de sua formalização. Como há uma grande possibilidade de estas regras ainda não estarem definidas em diversos cenários da MDA, aconselhamos a quem instanciar a MDA-Veritas fazer uma complementação desse procedimento delegando ao ator *Provedor da Solução*.

2. *Estender o metamodelo semântico.* Esta atividade envolve conhecimentos de metamodelagem e de semântica formal. Além do mais, o *Provedor da Solução* deve estar atento para metamodelos existentes que instanciem modelos formais nas plataformas onde os modelos semânticos poderão ser concretizados. É uma das atividades mais complexas da MDA-Veritas, porém se for delegada à um framework existente com uma linguagem formalizada de entrada através de um metamodelo, poderá ser bastante simplificada como aconteceu em nossa instanciação através do metamodelo disponível para a ferramenta Maude.

3. *Descrever as equações semânticas.* É uma atividade essencial da MDA-Veritas que deve ser baseada na técnica formal que define os modelos semânticos para que esta seja confiável. As *equações semânticas* respondem pelos principais questionamentos de eficácia da MDA-Veritas, pois são transformações que sustentarão a verificação de outras transformações através de mapeamentos de construções sintáticas em domínios semânticos. Contudo, desde as formulações mais primitivas da semântica denotacional, até onde é do nosso conhecimento, não há outra maneira de se dar esse mapeamento se não for através de transformações entre modelos ou código, não acrescentando ônus em esforços a quem instanciar a MDA-Veritas.

4. *Especificar processamento dos modelos semânticos.* É a atividade que caracteriza a semântica operacional dos modelos envolvidos nas transformações. Assim como a semântica estática, é uma tarefa que pode ser sabiamente delegada à plataforma de execução dos modelos semânticos. A depender dos modelos específicos de plataforma envolvidos e de suas variações, um número de regras de reescrita deve ser provido para diversos cenários de execução. Esta atividade então requisita que o *Provedor da Solução* esteja em sintonia com o *Cliente MDA* e com seus cenários de aplicação dos modelos em ambientes reais.

5. *Expressar a equivalência dos modelos.* É a atividade principal onde estabelece-se o mecanismo final de comparação de semântica dos modelos. Este mecanismo deverá ser construído pelo *Provedor da Solução* de acordo com elicitações das propriedades a serem preservadas junto ao *Cliente MDA*. A aplicação do *Verificador Formal* é condicionada à aplicação das atividades anteriores, podendo esta fazer uso de conceitos produzidos pelas demais, tais como as regras de processamento ou os domínios semânticos. A instanciação do *Verificador Formal* exige conhecimentos do *Provedor da Solução* de técnicas tais como model-checking, provas de teoremas, bissimulação e outras mais capazes de prover a resposta adequada. Desta forma, esta atividade caracteriza-se como o principal esforço na adoção da MDA-Veritas devido à ela não existir em processos comuns de compilação, onde características de modelos e códigos não necessitam de uma comparação de equivalência explícita.

8.1 Contribuições e Trabalhos Futuros

No Capítulo 1.5, apresentamos uma série de contribuições que esperávamos obter com a realização desse trabalho. A seguir, dissertamos resumidamente sobre o valor de cada contribuição obtida:

- *Incorporar técnicas formais à MDA*. Conseguimos apresentar uma visão simplificada para a comunidade MDA de diversos conceitos até então extremamente complexos da semântica formal: equações semânticas, semântica de dinâmica ou de execução, teoria das categorias e outros mais. Além de reusarmos diversos conceitos já existentes na MDA, também conseguimos uma suavização do uso de técnicas complexas como verificação de modelos e verificação de equivalência.

- *Melhorar confiabilidade na manipulação e processamento de PIMs e PSMs*. Ao reusarmos uma estratégia formal para derivarmos modelos, oferecemos um incentivo ao reuso de muitas teorias bem formuladas na ciência da computação que, devido ao pragmatismo apresentado pela engenharia de software, terminam por ficar esquecidos sem o uso merecido do seu potencial.

- *Melhorar confiabilidade na geração de código executável*. Os modelos semânticos descritos para PSMs apresentam características bem próximas da semântica de execução de códigos em linguagens de programação para plataformas específicas. Por exemplo, a formalização da *semântica interleaving* descrita é característica de códigos na linguagem C para plataformas PIC. Já a formalização da *semântica maximal step* descrita é característica de códigos na linguagem VHDL para imperativas a serem executadas em diversas plataformas de sistemas concorrentes. A formalização do comportamento desses modelos permitirá a representação correta dessas construções em linguagens específicas [Gomes and Costa, 2006], como por exemplo ANSI C, System C ou VERILOG, para diversos tipos de plataformas.

- *Aumentar o grau de representatividade semântica de plataformas de execução*. Vimos que diversos conceitos de plataformas específicas foram representados e considerados no desenvolver do trabalho. Isto enriqueceu bastante o poder de análise e a aplicabilidade da solução. Além do mais, o trabalho de posdoc definido em [MDA-VERITAS, 2011], prevê uma exploração mais detalhada dessa questão.

- *Permitir uma infraestrutura para verificação de preservação de semântica em MDA*. A arquitetura MDA-Veritas instanciada para sistemas concorrentes está pronta para o uso, e há diversos ensaios e idéias emergentes em outros paradigmas de modelagem e de programação.

- *Prover uma avaliação completa para o paradigma concorrente através de modelos em redes de Petri e redes IOPT*. Realizamos experimentos com a instanciação definida para a MDA-Veritas e pudemos identificar pontos fortes e pontos críticos da solução de acordo com o conjunto de ferramentas implementados e escolhidos. Com isso, pontos específicos da instanciação podem ser alterados visando ganhos nesses pontos críticos, além de que essa avaliação poderá guiar outras avaliações em diferentes paradigmas de instanciação onde diversos questionamentos ainda estejam obscuros para o *Provedor da Solução*.

As principais limitações recorrem a um problema comumente encontrado na comunidade MDA: a dificuldade em conseguir clientes interessados em aprender e aplicar a solução. Embora tenhamos conseguido algumas colaborações interessantes, a idéia da dificuldade de se aplicar esse tipo de técnicas afastou, por algumas vezes, outras colaborações que teriam um perfil de muito interesse ao nosso projeto. Além do mais, não foi possível, devido à escassez de mão de obra, prover uma solução finalizada sob o ponto de vista de usabilidade para o usuário final.

Como trabalhos futuros, é recomendado explorar mais aplicações na indústria para os conceitos dos *modelos semânticos* com detalhes específicos de plataforma, como os apresentados aqui, no âmbito do projeto de software para sistemas embarcados. Entre as principais linhas para pesquisas futuras, podemos citar:

1. Desenvolvimento de metamodelos semânticos com as características específicas de cada plataforma.

2. Incorporação de ações e definições semânticas para que sejam processadas em máquinas virtuais específicas como em um processo de compilação.

3. Desenvolvimento de um ambiente para integração dos mundos independentes e dependentes de plaforma quando lidando com modelos semânticos.

4. Equações semânticas mais robustas adequadas à realidade dos modelos industriais.

5. Adaptação a situações envolvendo mais de um modelo de entrada e mais de um modelo de saída.

6. Reuso de outras plataformas para engenharia semântica que se adequem aos propósitos do projeto. Por exemplo, atualmente tem-se grande interesse em ver como a ferramenta recente PLT Redex [Felleisen et al., 2009] se comportaria em substituição ao Maude.

Referências

[Aho et al., 1988] Aho, A. V., Sethi, R., and Ullman, J. D. (1988). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

[Alves, 2006] Alves, A. e. a. (2006). Web Services Business Process Execution Language Version 2.0. *OASIS Committee Draft - www.oasis-open.org*.

[Andrews et al., 2003] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business process execution language for web services version 1.1. Technical report, BEA, IBM, Microsoft, SAP, Siebel.

[Atlantic Zoo, 2011] Atlantic Zoo, T. (2011). The Atlantic Zoo. http://apps.eclipse.org/gmt/am3/zoos/atlanticZoo/.

[Barbosa et al., 2011] Barbosa, P., Barros, J. P., Ramalho, F., Gomes, L., Figueiredo, J., Moutinho, F., Costa, A., and Aranha, A. (2011). SysVeritas: A Framework for Verifying IOPT Nets and Execution Semantics within Embedded Systems Design. In *Technological Innovation for Sustainability*.

[Barbosa et al., 2009a] Barbosa, P., Costa, A., Ramalho, F., Figueiredo, J., Gomes, L., and Junior, A. (2009a). Checking Semantics Equivalence of MDA Transformations in Concurrent Systems. *Journal of Universal Computer Science (J.UCS)*. http://www.jucs.org/jucs.

[Barbosa et al., 2009b] Barbosa, P., Costa, A., Ramalho, F., Figueiredo, J., Gomes, L., and Junior, A. (2009b). Modeling Complex Petri Nets Operations in the Model-Driven Architecture. In *IEEE IECON'09*.

[Barbosa et al., 2010a] Barbosa, P., Costa, A., Ramalho, F., Figueiredo, J., Gomes, L., and Junior, A. (2010a). A MDA-based Contribution for Integrating Web Services within Embedded System's Design. In *Proceedings of INDIN (8th IEEE International Conference on Industrial Informatics)*.

[Barbosa et al., 2010b] Barbosa, P., Ramalho, F., Figueiredo, J., Costa, A., Gomes, L., and Junior, A. (2010b). Semantic Equations for Formal Models in the Model-Driven Architecture. In *Emerging Trends in Technological Innovation*.

[Barbosa et al., 2008a] Barbosa, P., Ramalho, F., Figueiredo, J., and Junior, A. (2008a). An Extended mda Architecture for Ensuring Semantics-Preserving Transformations. In *Proceedings of 32nd Annual IEEE Software Engineering Workshop*.

[Barbosa et al., 2008b] Barbosa, P. E. S., Ramalho, F., Figueiredo, J. C. A., and Junior, A. D. (2008b). Incorporating Semantic Algebra in the MDA Framework. In *Proceedings of the Third International Conference on Software and Data Technologies (ICSOFT). Special Session on Metamodelling - Utilization in Software Engineering*, pages 330–336.

[Baresi et al., 2006] Baresi, L., Ehrig, K., and Heckel, R. (2006). Verification of Model Transformations: A Case Study with Bpel. In Montanari, U., Sannella, D., and Bruni, R., editors, *TGC*, volume 4661 of *Lecture Notes in Computer Science*, pages 183–199. Springer.

[Barr, 2011] Barr, M. (2011). Embedded systems glossary. http://www.netrino.com/Embedded-Systems/Glossary.

[Basili et al., 1994] Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley.

[Best et al., 1997] Best, E., Esparza, J., Grahlmann, B., Melzer, S., Römer, S., and Wallner, F. (1997). The pep verification system. In *FEmSys*.

[Bettin, 2004] Bettin, J. (2004). Model-Driven Software Development: An emerging paradigm for industrialized software asset development. http://www.softmetaware.com/mdsd-and-isad.pdf.

[Bezivin et al., 2003] Bezivin, J., Breton, E., Valduriez, P., and Dupr, G. (2003). The ATL Transformation-Based Model Management Framework. Technical report, IRIN.

[Bézivin et al., 2006] Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., and Lindow, A. (2006). Model Transformations? Transformation Models! pages 440–453.

[Boronat and Meseguer, 2008] Boronat, A. and Meseguer, J. (2008). An algebraic semantics for mof. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 377–391, Berlin, Heidelberg. Springer-Verlag.

[Boronat and Meseguer, 2009] Boronat, A. and Meseguer, J. (2009). Moment2: Emf model transformations in maude. In Vallecillo, A. and Sagardui, G., editors, *JISBD*, pages 178–179.

[Bruni et al., 2001] Bruni, R., Meseguer, J., Montanari, U., and Sassone, V. (2001). Functorial Models for Petri nets. *Inf. Comput.*, 170(2):207–236.

[Budinsky et al., 2003] Budinsky, F., Brodsky, S. A., and Merks, E. (2003). *Eclipse Modeling Framework*. Pearson Education.

[Cabot et al., 2010] Cabot, J., Clarisó, R., Guerra, E., and de Lara, J. (2010). Verification and validation of declarative model-to-model transformations through invariants. *J. Syst. Softw.*, 83:283–302.

[Chalub and Braga, 2005] Chalub, F. and Braga, C. (2005). An Implementation of Modular SOS in Maude. In *Master's thesis, Universidade Federal Fluminense*.

[Chen et al., 2005] Chen, K., Sztipanovits, J., Abdelwahed, S., and Jackson, E. K. (2005). Semantic Anchoring with Model Transformations. In *ECMDA-FA*, pages 115–129.

[Clark and Warmer, 2002] Clark, T. and Warmer, J., editors (2002). *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*. Springer.

[Clavel et al., 2011] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2011). Maude Manual (Version 2.6).

[Clavel and Meseguer, 1997] Clavel, M. and Meseguer, J. (1997). Reflection in rewriting logic and its applications in the maude language. In *In IMSA'97, pages 128–139. Information-Technology Promotion Agency*, pages 128–139.

[Costa, 2010] Costa, A. (2010). *Petri Net Model Decomposition - A Model Based Approach Supporting Distributed Execution*. PhD thesis, Universidade Nova de Lisboa.

[Costa et al., 2010] Costa, A., Barbosa, P., Gomes, L., Ramalho, F., Figueiredo, J., and Junior, A. (2010). Properties Preservation in Distributed Execution of Petri Nets Models. In *Emerging Trends in Technological Innovation*.

[Csertan et al., 2002] Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., and Varro, D. (2002). VIATRA: Visual automated transformations for formal verification and validation of UML models.

[Czarnecki and Helsen, 2003] Czarnecki, K. and Helsen, S. (2003). Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop On Generative Techniques in the Context of the Model Driven Architecture*.

[David, 1995] David, R. (1995). Grafcet: A Powerful Tool for Specification of Logic Controllers. *IEEE Transactions on Control System Technology*, 3(3).

[David and Alla, 1992] David, R. and Alla, H. (1992). *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Desel et al., 2001] Desel, J., Juhás, G., and Eichstätt, K. U. (2001). What is a petri net? informal answers for the informed reader. In *Unifying Petri Nets, LNCS 2128*, pages 1–27. Springer.

[Durán and Meseguer, 2000] Durán, F. and Meseguer, J. (2000). Parameterized theories and views in full maude 2.0. *Electr. Notes Theor. Comput. Sci.*, 36:316–338.

[Edmund M. Clarke et al., 1999] Edmund M. Clarke, J., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press.

[Ehrig et al., 2007] Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., and Taentzer, G. (2007). Information Preserving Bidirectional Model Transformations. In Dwyer, M. B. and Lopes, A., editors, *FASE*, volume 4422 of *Lecture Notes in Computer Science*, pages 72–86. Springer.

[Ehrig et al., 2006a] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006a). *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 1 edition.

[Ehrig et al., 2006b] Ehrig, H., Hoffmann, K., and Padberg, J. (2006b). Transformations of petri nets. *Electron. Notes Theor. Comput. Sci.*, 148:151–172.

[Eker et al., 2003] Eker, S., Meseguer, J., and Sridharanarayanan, A. (2003). The Maude LTL Model Checker and its Implementation. In *SPIN'03: Proceedings of the 10th international conference on Model checking software*, pages 230–234, Berlin, Heidelberg. Springer-Verlag.

[Engels et al., 2000] Engels, G., Hausmann, J. H., Heckel, R., and Sauer, S. (2000). Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In Evans, A., Kent, S., and Selic, B., editors, *UML 2000 - The Unified Modeling*

Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings, volume 1939 of *LNCS*, pages 323–337. Springer.

[Fabbri et al., 1995] Fabbri, S. C. P. F., Maldonado, J. C., Masiero, P. C., Delamaro, M. E., and Wong, W. E. (1995). Mutation testing applied to validate specifications based on petri nets. In *Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques VIII*, pages 329–337.

[Falleri et al., 2006] Falleri, J.-r., Huchard, M., and Nebut, C. (2006). Towards a Traceability Framework for Model Transformations in Kermeta. In *In: ECMDA-TW Workshop*.

[Farwer and Leuschel, 2004] Farwer, B. and Leuschel, M. (2004). Model Checking Object Petri nets in Maude and Prolog. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 20–31.

[Felleisen et al., 2009] Felleisen, M., Findler, R. B., and Flatt, M. (2009). *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition.

[France and Bieman, 2001] France, R. and Bieman, J. (2001). Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In *In Proceedings of Internacional Conference on Software maintenance (ICSM 2001)*.

[Gentleware, 2011] Gentleware (2011). Poseidon for uml v. 4. http://www.gentleware.com.

[Gheyi et al., 2005] Gheyi, R., Massoni, T., and Borba, P. (2005). A Rigorous Approach for Proving Model Refactorings. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 372–375, New York, NY, USA. ACM.

[Gheyi et al., 2007] Gheyi, R., Massoni, T., and Borba, P. (2007). A Static Semantics for Alloy and its Impact in Refactorings. *Electron. Notes Theor. Comput. Sci.*, 184:209–233.

[Gomes et al., 2007] Gomes, L., Barros, J. P., Costa, A., and Nunes, R. (2007). The Input-Output Place-Transition Petri Net Class and Associated Tools. In *Proceedings of the 5th IEEE International Conference on Industrial Informatics*, pages 23–26. IEEE Computer Society Press.

[Gomes et al., 2005] Gomes, L., Barros, J. P., Costa, A., Pais, R., and Moutinho, F. (2005). Formal methods for embedded systems co-design: the fordesign project. In *ReCoSoC?05- Reconfigurable Communication-centric Systems-on-Chip - Workshop Proceedings*.

[Gomes and Costa, 2006] Gomes, L. and Costa, A. (2006). Petri nets as Supporting Formalism within Embedded Systems Co-design. In *SIES-2006 - 2006 IEEE International Symposium on Industrial Embedded Systems*.

[Gomes and Costa, 2007] Gomes, L. and Costa, A. (2007). Petri net Splitting Operation within Embedded Systems Co-design. In *Proceedings of INDIN (5th IEEE International Conference on Industrial Informatics)*.

[Hausmann, 2005] Hausmann, J. (2005). *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Techniques*. PhD thesis, University of Paderborn.

[Holzmann, 1997] Holzmann, G. J. (1997). The model checker SPIN. *Software Engineering*, 23(5):279–295.

[Hülsbusch et al., 2010] Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., and Wehrheim, H. (2010). Full Semantics Preservation in Model Transformation - A Comparison of Proof Techniques. Technical Report TR-CTIT-10-09, Enschede.

[Hülsbusch et al., 2010] Hülsbusch, M., König, B., Rensink, A., Semenyak, M., and Nederpel, R. (2010). Full semantics preservation in model transformation - a comparison of proof techniques. *Technical Report*, TR-CTT-10-09.

[Jackson, 2002] Jackson, D. (2002). Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2).

[Jensen, 1986] Jensen, K. (1986). Computer tools for construction, modification and analysis of petri nets. *One-Day Seminar at the Bocconi University of Milan on: Applicability of Petri Nets to Operations Research*, pages 68–94. NewsletterInfo: 25.

[Jensen, 1991] Jensen, K. (1991). Coloured petri nets: A high level language for system design and analysis. *Lecture Notes in Computer Science; Advances in Petri Nets 1990*, 483:342–416. NewsletterInfo: 39.

[Josuttis, 2007] Josuttis, N. M. (2007). *SOA in Practice: The Art of Distributed System Design*. O'Reilly, Beijing.

[Katoen, 1999] Katoen, J. P. (1999). *Concepts, Algorithms, and Tools for Model Checking*. Lectures Notes of the Course "Mechanised Validation of Parallel Systems. Friedrich-Alexander Universitat Erlangen-Nurnberg.

[Kent et al., 1999] Kent, S., Gaito, S., and Ross, N. (1999). A Meta-model Semantics for Structural Constraints in UML. In Kilov, H., Rumpe, B., and Simmonds, I., editors, *Behavioral specifications for businesses and systems*, chapter 9, pages 123–141. Kluwer Academic Publishers, Norwell, MA.

[KentModellingFramework, 2011] KentModellingFramework (2011). Kent ocl library. http://www.cs.kent.ac.uk/projects/kmf/.

[KlasseObjecten, 2011] KlasseObjecten (2011). Octopus: Ocl tool for precise uml specifications. http://www.klasse.nl/octopus/index.html.

[Kleppe and Warmer, 2003] Kleppe, A. and Warmer, J. (2003). Do MDA Transformations Preserve Meaning? An Investigation into Preserving Semantics. In *Proceedings of the 1st International Workshop on Metamodelling for MDA*.

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Professional.

[Lawvere and Schanuel, 1997] Lawvere, F. and Schanuel, S. (1997). *Conceptual Mathematics: A First Introduction to Categories*. Cambridge: Cambridge University Press.

[Li et al., 2005] Li, H., Thompson, S., and Reinke, C. (2005). The Haskell Refactorer, HaRe, and its API. *Electron. Notes Theor. Comput. Sci.*, 141(4):29–34.

[Lohmann, 2007] Lohmann, N. (2007). A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In Hee, K. v., Reisig, W., and Wolf, K., editors, *Proceedings of the Workshop on Formal Approaches to Business Processes and Web Services (FABPWS'07)*, pages 21–35. University of Podlasie.

[M. Clavel and Meseguer, 2000] M. Clavel, S. Eker, P. L. and Meseguer, J. (2000). Principles of Maude. In Meseguer, J., editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers.

[Madl et al., 2006] Madl, G., Abdelwahed, S., and Schmidt, D. C. (2006). Verifying Distributed Real-Time Properties of Embedded Systems via Graph Transformations and Model Checking. *Real-Time Syst.*, 33(1-3):77–100.

[Manes, 1975] Manes, E. G., editor (1975). *Category Theory Applied to Computation and Control, Proceedings of the First International Symposium, San Francisco, CA, USA, February 25-26, 1974, Proceedings*, volume 25 of *Lecture Notes in Computer Science*. Springer.

[Mathur and Wong, 1994] Mathur, A. P. and Wong, W. E. (1994). An empirical comparison of data flow and mutation-based test adequacy criteria. *Softw. Test., Verif. Reliab.*, 4(1):9–31.

[Mazidi et al., 2005] Mazidi, M. A., Mazidi, J., McKinlay, R., and Ingendorf, P. (2005). *PIC Microcontroller and Embedded Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[MDA-VERITAS, 2011] MDA-VERITAS (2011). The MDA-Veritas Project. http://code.google.com/p/mdaveritas/.

[Meseguer and Braga, 2004] Meseguer, J. and Braga, C. (2004). Modular Rewriting Semantics of Programming Languages. In *In Proceedings of the 10th International Conference, AMAST'04*, pages 364–378. Springer.

[Meseguer and Montanari, 1988] Meseguer, J. and Montanari, U. (1988). Petri Nets are Monoids: a new Algebraic Foundation for net Theory. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science (LICS 1988)*, pages 155–164. IEEE Computer Society Press.

[Meseguer and Rosu, 2004] Meseguer, J. and Rosu, G. (2004). Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools. In *IJCAR*, pages 1–44.

[Meseguer and Rosu, 2007] Meseguer, J. and Rosu, G. (2007). The Rewriting Logic Semantics Project. *Theor. Comput. Sci.*, 373(3):213–237.

[Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). MDA Guide Version 1.0.1. In *Object Management Group (OMG)*.

[Mokhati et al., 2007] Mokhati, F., Gagnon, P., and Badri, M. (2007). Verifying UML Diagrams with Model Checking: A Rewriting Logic Based Approach. In *QSIC*, pages 356–362. IEEE Computer Society.

[Mosses, 1975] Mosses, P. D. (1975). The Semantics of Semantic Equations. In *Proceedings of the 3rd Symposium on Mathematical Foundations of Computer Science*, pages 409–422, London, UK. Springer-Verlag.

[Moutinho et al., 2011] Moutinho, F., Gomes, L., Barbosa, P., Barros, J. P., Ramalho, F., Figueiredo, J., Costa, A., and Aranha, A. (2011). Petri net based Specification and Verification of Globally-Asynchronous-Locally-Synchronous Systems. In *Technological Innovation for Sustainability*.

[Moutinho et al., 2010] Moutinho, F., Gomes, L., Ramalho, F., Figueiredo, J., Barros, J., Barbosa, P., Pais, R., and Costa, A. (2010). Ecore Representation for Extending PNML for Input-Output Place-Transition Nets. In *IEEE IECON'10*.

[Murata and Koh, 1980] Murata, T. and Koh, J. Y. (1980). Reduction and expansion of live and safe marked graphs. *IEEE Trans. on Circuits and Systems*, Cas-27(1):68–70. NewsletterInfo: 6.

[Musuvathi et al., 2002] Musuvathi, M., Park, D., Chou, A., Engler, D. R., and Dill, D. L. (2002). CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*.

[Narayanan and Karsai, 2006] Narayanan, A. and Karsai, G. (2006). Using Semantic Anchoring to Verify Behavior Preservation in Graph Transformations. *Electronic Communications of the EASST*, 4(2006).

[Narayanan and Karsai, 2008] Narayanan, A. and Karsai, G. (2008). Towards Verifying Model Transformations. *Electron. Notes Theor. Comput. Sci.*, 211:191–200.

[Neuhausser and Noll, 2007] Neuhausser, M. and Noll, T. (2007). Abstraction and Model Checking of Core Erlang Programs in Maude. *Electron. Notes Theor. Comput. Sci.*, 176:147–163. http://dx.doi.org/10.1016/j.entcs.2007.06.013.

[Ogata and Futatsugi, 2007] Ogata, K. and Futatsugi, K. (2007). Comparison of Maude and SAL by Conducting Case Studies Model Checking a Distributed Algorithm. *IEICE Transactions*, 90-A:1690–1703.

[OMG, 2005] OMG (2005). *MOF QVT Final Adopted Specification*. Object Modeling Group.

[OMG, 2011a] OMG (2011a). Model-Driven Architecture. http://www.omg.org/mda/.

[OMG, 2011b] OMG (2011b). Object Management Group. http://www.omg.org.

[OMG, 2011c] OMG (2011c). Ocl Specification Page. http://www.omg.org/spec/OCL.

[Pais et al., 2005] Pais, R., Barros, J., and Gomes, L. (2005). A Tool for Tailored Code Generation from Petri Net Models. In *10^{th} IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005).*

[Petri, 1962] Petri, C. A. (1962). *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[Petter et al., 2009] Petter, A., Behring, A., and Mühlhäuser, M. (2009). Solving constraints in model transformations. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 132–147, Berlin, Heidelberg. Springer-Verlag.

[Podnieks, 2005] Podnieks, K. (2005). Mda: Correctness of Model Transformations. Which Models are Schemas? In *Proceedings of 6th International Baltic Conference on Databases and Information Systems*, volume 118, pages 185–197. IOS Press.

[Ray and Sumners, 2007] Ray, S. and Sumners, R. (2007). Combining Theorem Proving with Model Checking through Predicate Abstraction. In *IEEE Des. Test*, pages 132–139. IEEE Computer Society Press.

[Rensink and Nederpel, 2008] Rensink, A. and Nederpel, R. (2008). Graph Transformation Semantics for a QVT Language. *Electron. Notes Theor. Comput. Sci.*, 211:51–62.

[Romero et al., 2007] Romero, J. R., Rivera, J. E., Durán, F., and Vallecillo, A. (2007). Formal and tool support for model driven engineering with Maude. *Journal of Object Technology*, 6(9).

[Ruscio et al., 2006] Ruscio, D. D., Jouault, F., Kurtev, I., Bezivin, J., and Pierantonio, A. (2006). Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report 06.02, LINA.

[Schmidt, 1986] Schmidt, D. A. (1986). *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA.

[Scott and Strachey, 1971] Scott, D. and Strachey, C. (1971). Towards a Mathematical Semantics for Computer Languages. In *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*.

[Stahl, 2005] Stahl, C. (2005). A Petri Net Semantics for BPEL. (Technical Report 188, Humboldt-Universität zu Berlin).

[Stahl et al., 2006] Stahl, T., Voelter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

[Stehr and Csaba, 2001] Stehr, M.-O. and Csaba, P. (2001). Rewriting Logic as a Unifying Framework for Petri nets. *Unifying Petri Nets, LNCS*, pages 250–303. Springer.

[Stehr et al., 2001] Stehr, M.-O., Meseguer, J., and Ölveczky, P. C. (2001). Rewriting Logic as a Unifying Framework for Petri Nets. In *In Unifying Petri Nets, Advances in Petri Nets*, pages 250–303.

[Tennent, 1976] Tennent, R. D. (1976). The Denotational Semantics of Programming Languages. *Commun. ACM*, 19(8):437–453.

[Troya and Vallecillo, 2010] Troya, J. and Vallecillo, A. (2010). Towards a rewriting logic semantics for atl. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 230–244, Berlin, Heidelberg. Springer-Verlag.

[Weber and Kindler, 2003] Weber, M. and Kindler, E. (2003). The petri net markup language. pages 124–144.