Rui Gonçalves Domingues

Mestrado em Engenharia Informática

Selecção de Testes a partir de Redes de Petri Algébricas: um método e uma ferramenta

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Prof. Dr. Vasco Amaral, Prof. Auxiliar, FCT/UNL

Júri

Presidente: Prof. Dr. João Leite
Arguente: Profª. Dra. Ana Paiva
Vogal: Prof. Dr. Vasco Amaral

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2012

Selecção de Testes a partir de Redes de Petri Algébricas: um método e uma ferramenta

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Para os meus extremosos pais

Agradecimentos

No decurso da elaboração desta tese de mestrado contei com a colaboração de diferentes pessoas e organizações, às quais gostaria de agradecer.

Primeiramente ao Professor Vasco Amaral pela alavancagem e interesse em me manter na investigação sobre o tema afecto a esta dissertação, pela gestão burocrática e pelos procedimentos necessários para que na maior parte do tempo eu fosse remunerado pelos trabalhos. Agradeço especialmente pela força e incentivo que me foi dando nos momentos mais desesperantes e de dúvida quanto ao sucesso da mesma.

Ao Dr. Levi Lúcio pelas diversos encontros/reuniões e discussões que tiveram lugar no preâmbulo do desenvolvimento desta dissertação, cujo tema foi por este coproposto por derivar maioritariamente de trabalhos por ele efectuados no decorrer da sua tese de doutouramento.

Quero agradecer ainda ao Dr. Bruno Barroca, pois foi quem mais esteve presente durante todo o desenvolvimento. Um obrigado pela disponibilidade para longas discussões teóricas e técnicas, pelo suporte dado praticamente a qualquer hora do dia, pelas interrogações levantadas durante os trabalhos que levaram à existência das soluções agora propostas. Agradeço ainda ao colega Cláudio Gomes pelo companheirismo já nos últimos meses de desenvolvimento e pela disponibilidade para esclarecer alguma dúvida sobre o funcionamento de ferramentas utilizadas. Na mesma linha, e também já nos últimos momentos, agradeço o companheirismo de Ankica Barisic.

O meu agradecimento vai ainda para a minha família que sempre me apoiou no meu percurso académico. À minha irmã pelo incentivo e estímulo nas alturas mais críticas. Aos meus pais um agradecimento especial pelo carinho, motivação, apoio financeiro numa grande parte da minha jornada académica e por acreditarem sempre em mim. À Alexandrina pela paciência, pela compreensão, pelo suporte de todos os dias, pelos afectos e ainda pela capacidade de me fazer acreditar nas minhas capacidades.

Dignos de nota são também os colegas mais próximos, que partilharam o seu percurso académico com o meu, pelo entusiasmo com que levámos o curso até ao fim, pelas longas noites de convivência em trabalho, e pela inter-ajuda emocional e laboral.

O meu obrigado, sem qualquer ordem preferencial, ao Danilo Manmohanlal, Nuno Luís, João Cartaxo, Pedro Bernardo, Nuno Boavida, Emanuel Couto, Pedro Sousa, Filipe Grangeiro, Hélio Dolores, André Morão, Gonçalo Martins, Ricardo Martins, José Correia, Bruno Santos, Leonardo Jardim, André Vicente e ao David Costa.

Agradeço ainda ao Departamento de Informática pela facultação das instalações e pelo financiamento facultado em diferentes períodos do desenvolvimento da dissertação, assim como à Fundação para Ciência e Tecnologia pelo mesmo motivo.

E por fim a todos os meus amigos que sempre estiveram presentes e a todas as pessoas tais como o pessoal do secretariado que, direta ou indiretamente, contribuíram para a execução desta dissertação de mestrado.

Sumário

A abordagem de testes baseados em modelos (MBT) deriva de forma directa do aproveitamento das metodologias orientadas por modelos e da sua aplicação à disciplina de Teste de software. MBT consiste num processo cujas fases são facilmente identificáveis como: criação de um modelo do sistema sob teste (SUT), que pode ser de diversos tipos e que se pretende abstracto em relação ao funcionamento do SUT, a geração de testes abstractos a partir do modelo e a sua concretização em testes executáveis sobre o sistema, e por fim a execução dos mesmos, a recolha de resultados e respectiva análise.

Nesta dissertação o interesse recai essencialmente sobre a geração de testes abstractos efectuada a partir de modelos comportamentais especificados em APN (Petri nets algébricas), e que modelam algum aspecto comportamental do SUT. No trabalho desta dissertação iremos recorrer a uma linguagem de especificação de padrões de teste - a SATEL (Semi-Automatic Testing Language) - para fazer a especificação das intenções de teste.

A SATEL consiste numa linguagem baseada em modelos, a qual está desenvolvida com uma semântica essencialmente denotacional. Esta possibilita ao engenheiro de teste utilizar o conhecimento que possui sobre o SUT, pois permite a especificação de intenções de teste sobre um subconjunto dos comportamentos possíveis do SUT. Mais se acrescenta que é uma linguagem cuja especificação é feita por definição em CO-OPN(Concurrent Object-Oriented Petri Nets).

Concluindo, com a presente dissertação pretende-se compôr as linguagens SATEL e APN, e desenvolver não só a semântica operacional desta composição como também a sua implementação através de uma ferramenta de geração de testes para especificações APN, contribuindo assim para reduzir a falta de soluções concretas e funcionais de geração automática de testes a partir da especificação de intenções de testes sobre tipos algébricos.

Palavras-chave: Teste baseado em modelos, Selecção de Testes, SATEL, Redes de Petri algébricas, Engenharia conduzida por modelos

Summary

Model-based testing (MBT) is a software testing approach which derives directly from object-oriented methodologies and its application to software testing. MBT consists on a process composed of some easily identifiable steps: creation of a abstract model of the system under test(SUT), which can be of different types, the derivation of abstract tests from model and its concretization into executable tests, and at last, its execution, and the result gathering for analysis. MBT is therefore a mean to identify system fails in order to correct them.

In this thesis, we are essentially interested, in the second phase of the process - the generation of abstract tests - performed from behavioral models specified in APN (Algebraic Petri nets), which model some operational aspect of SUT, being that in the this approach, it is used a test specification language - SATEL (Semi-Automatic TEsting Language) - to specify test intentions. SATEL consists in a model-based test language, which was mostly developed with a denotational semantic, and it allows the use of knowledge test engineers have about the SUT, once it allows the specification of test intentions over a subset of the possible behaviors of the SUT. More we can add, that it is a language developed to write tests for CO-OPN(Concurrent Object-Oriented Petri Nets) specifications. In this thesis we compose the languages SATEL and APN to develop not only an operational semantic of this composition, but also, to perform its implementation in a test generation tool for APN specifications.

Keywords: Model-Based Testing, Test selection, SATEL, Algebraic Petri-nets, Model-driven Engineering.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Listagens

1

Introdução

1.1 Introdução geral e motivação

A actividade de Teste de Software consiste essencialmente na identificação de falhas, erros e funcionalidades não expectáveis de um sistema, a qual pode ser levada a cabo em qualquer fase do ciclo de desenvolvimento de software, tendo em vista a adopção de medidas de correcção dos sistemas. É uma actividade importante para a verificação do software que complementa e é complementada com outras técnicas de verificação formal, exaustivas e pouco escaláveis como *Model Checking* ou *Theorem Proving*, que conjuntamente aferem a qualidade do sistema.

Com a crescente produção de sistemas de software, da sua respectiva complexidade e da necessidade de desenvolvimento automatizado, rápido, preciso e livre de falhas, o Teste Software, como parte integrante do processo de desenvolvimento, tem vindo a tomar um lugar de elevado relevo, visto que consiste em procedimentos que permitem não só validar o correcto funcionamento de um sistema, mas também identificar potenciais falhas e respectivas causas.

Com o já referido aumento de complexidade dos sistemas, o desenvovimento de software tem hoje que fazer face a esse aumento ao nível de domínio de problema, ferramentas e plataformas. Uma forma de atacar o problema do aumento da complexidade é incitar o desenvolvimento de software conduzido por modelos *(Model-Driven Development(MDD))* promovendo a utilização de modelos (a um nível de abstracção adequado) que capturem a complexidade essencial do problema de uma forma minimalista e ignorem a complexidade acidental que lhe é inerente. É neste sentido que

surge o Teste de Software Baseado em Modelos (MBT).

Derivar um conjunto de testes exaustivo, com todas as possibilidades de execução, é trivialmente aceite como inexequível, ou mesmo impossível na maioria das aplicações de software do mundo real. Por esse motivo o MBT tem como principal objectivo seleccionar testes a partir de modelos de uma forma controlada, utilizando conhecimento de alto nível (ao nível dos requisitos), permitindo assim reduzir o espaço de estados do sistema a pesquisar.

Na literatura científica podemos encontrar diferentes técnicas MBT que utilizam diferentes metodologias quanto à utilização dos modelos; em particular podem utilizar-se modelos do SUT (como por exemplo *task-models* [3]) e a partir desses gerar os testes, ou pode partir-se de modelos/padrões dos testes que servem para conduzir a geração dos testes e utilizar-se as especificações para obter os oráculos. É neste último tópico que esta dissertação se vai centrar.

Em suma, conclui-se que esta dissertação tem como motivação principal fazer face à falta de implementações de linguagens para a especificação de testes abstractos. Neste sentido pretende-se obter uma implementação de uma linguagem, única do estilo, que se baseia em especificações do SUT através de Redes de Petri Algébricas [22], a qual foi apenas conceptualizada através de uma semântica essencialmente denotacional, e que por esse motivo carece de validação, pondo em causa a própria exequibilidade do MBT. Essa linguagem é a SATEL [17].

1.2 Descrição e contextualização do problema

Na presente dissertação pretende desenvolver-se trabalho sobre a abordagem MBT, utilizando-se para esse efeito a linguagem meta-modelada SATEL para especificar as intenções de testes.

Aquando da génese da linguagem SATEL, esta foi desenhada tendo por base modelos especificados através do formalismo CO-OPN, com o qual se pode expressar comportamento de forma modular e concorrente de um sistema. A SATEL na qualidade de linguagem de especificação de testes tinha portanto, de ter obrigatoriamente um formalismo acoplado que permitisse especificar os SUT; e é neste sentido que vemos a SATEL como uma linguagem composta. No seguimento desta ideia, trabalhos foram feitos com o fim de explorar a capacidade de a SATEL ser composta com outros formalismos de especificação de sistemas, tal como o HALL (linguagem de especificação de interfaces gráficos sob uma abordagem MDD) [5], e é também no seguimento dessa ideia que nesta dissertação se pretende utilizar o formalismo das Redes de Petri Algébricas [22] para ser acoplado à linguagem SATEL, que consistem num formalismo de

especificação de sistemas concorrentes mais simples que o CO-OPN, quer em termos de especificação, quer de semântica.

Esta escolha pelas APN é justificada de acordo com as restrições temporais acometidas para realização desta dissertação, com o foco principal que é a geração de testes abstractos e não tanto a especificação dos modelos. Uma vez que a SATEL é uma linguagem originalmente desenvolvida sobre CO-OPN, que permite especificar modelos de sistemas concorrentes e reactivos, e sendo que as APN são entidades essencialmente de simulação, que não respondem a estímulos externos e nem permitem observação da resposta a esses estímulos, é necessário fazer alguma manipulação e estender a linguagem APN, para que possa prever estas duas capacidades. Designaremos doravante esta extensão por EAPN (*Extended-APN*).

Na especificação da linguagem foram definidas as sintaxes concreta e abstracta, bem como a sua semântica denotacional seguindo uma notação matemática e utilizando teoria de conjuntos. Esta forma de especificação semântica de linguagens é por um lado, propícia para a realização de provas formais e levantamento de propriedades gerais(teoremas) da linguagem no momento da sua conceptualização formal. No entanto, por outro lado, com este tipo de especificação matemática de linguagens a implementação não é derivável directamente, pois não existem linguagens de programação com o nível declarativo apropriado para implementar semânticas de linguagens directamente a partir de semânticas denotacionais formalizadas matematicamente. Disto discorre que a linguagem SATEL foi validada formalmente e portanto é coerente, porém não existe uma implementação para ela que permita efectivamente gerar testes.

Portanto, pretende-se obter uma implementação da semântica operacional da SATEL composta com as EAPN, representada como SATEL$\oplus$EAPN, que nos permita validar a linguagem SATEL quanto à capacidade efectiva de gerar testes abstractos de uma forma fortemente conexa ao que foi teorizado aquando da sua definição.

1.3 Resumo da solução proposta

No âmbito desta dissertação há o interesse, como já referido, em desenvolver uma implementação da semântica operacional para o SATEL$\oplus$EAPN com o fim de gerar testes abstractos a partir de modelos de sistemas especificados em EAPN. Não foi mencionado anteriormente mas, para além de existir interesse nas APN por permitirem descrever sistemas concorrentes, há um interesse acrescido na utilização das EAPN, no sentido em que futuramente se pretende integrar a ferramenta de teste com o *model checker* AlPiNa[1]. Esta ferramenta, desenvolvida pelo grupo SMV da Universidade de

[1]URL:http://alpina.unige.ch

Genebra, permitir-nos-á obter os modelos já verificados, isentos de propriedades indesejáveis, como *deadlocks* ou corridas, e que garantam outras como *reachibility* de determinados estados. Deste modo a ferramenta de teste deverá receber os modelos já eficientemente verificados.

Tendo em vista a geração de testes abstractos a partir de modelos SATEL⊕EAPN, já anotados com o respectivo oráculo, propõem-se quatro tarefas preliminares que dão alavancagem à solução a apresentar. São elas:

1. Propôr uma extensão às APN (EAPN), de forma a que a sua composição com a SATEL seja possível;

2. Compôr a linguagem SATEL com a das EAPN;

3. Implementar um editor gráfico para a linguagem composta, com o qual se possa criar os modelos com que se irá trabalhar.

4. Implementação do motor de geração de testes

Uma vez que os tópicos desta dissertação acentam sobre metodologias conduzidas por modelos, as ferramentas a utilizar serão também elas orientadas a modelos. Ir-se-á utilizar ferramentas assentes na *Eclipse Model Framework*(EMF) que nos permitem trabalhar com modelos *UML-Based*, e com ferramentas desenvolvidas no grupo de investigação em que este trabalho se insere. Deste modo o editor sugerido no ponto 3 será implementado recorrendo à ferramenta *EMFText*, que permite, rapidamente, através de uma gramática *EBNF-like*, prototipar um editor textual. Esta ferramenta utilizará os metamodelos previamente desenvolvidos na plataforma EMF.

Na posse de modelos SATEL⊕EAPN, a solução proposta passa por submeter esses modelos a um *pipelining* transformacional, até alcançar os resultados finais. Na figura 1.1 podemos observar essa situação.

Primeiramente, os modelos SATEL⊕EAPN passam por um pré-processamento para tornar as fases de transformação mais fáceis e convenientes às transformações subsequentes, nomeadamente são feitas alterações aos nomes das variáveis e *unfolding* de algumas estruturas complexas. Na etapa 2 os modelos pré-processados são refinados com ATL, tendo em vista a inicialização de algumas estruturas dos modelos, que ajudarão a fazer face a deficiências existentes entre as estruturas do *Eclipse Modeling Framework* e a ferramenta de transformação. Na etapa 3 dá-se então a transformação mais importante de todo o processo, em que é feita uma transformação dos modelos em SATEL⊕EAPN para modelos MPrologTR. Nesta transformação identificam-se padrões bem específicos, existentes nos modelos SATEL⊕EAPN, visando transformá-los em padrões de um metamodelo da linguagem Prolog. Estes padrões corresponderão

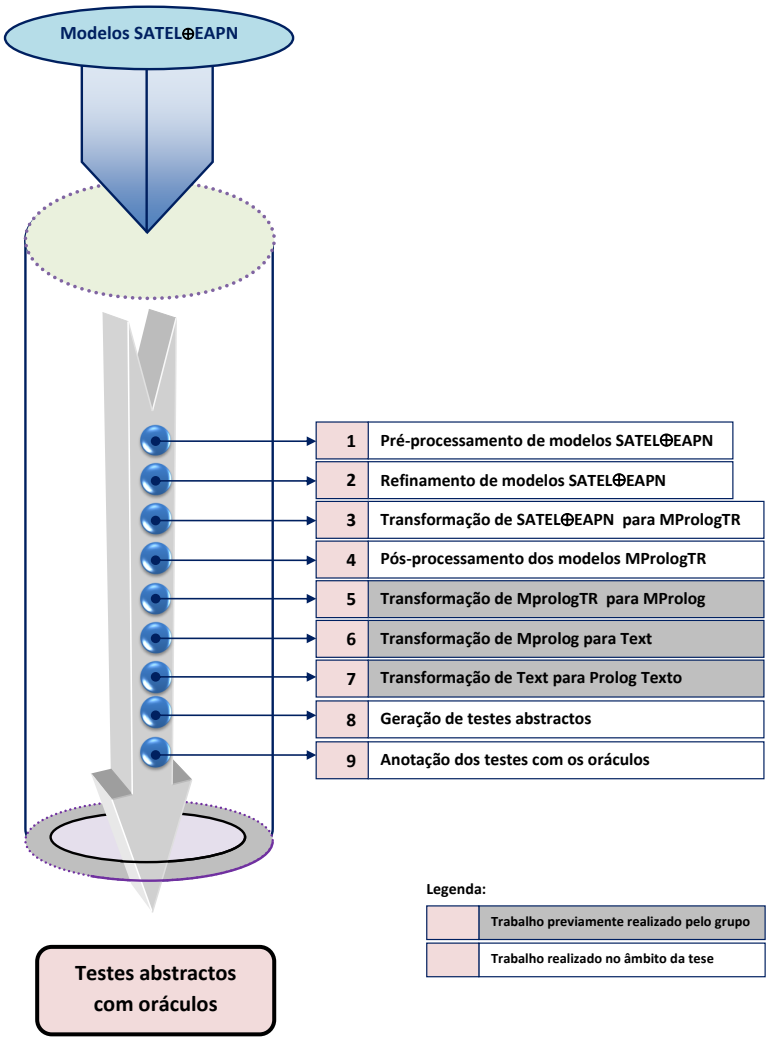

Figura 1.1: Visão geral da solução proposta

a cláusulas que poderão ser conferidas ("*queried*") com vista à obtenção de soluções, que neste caso serão os testes pretendidos.

O metamodelo MPrologTR (MProlog Term Reference) é um metamodelo em que é possível referenciar variáveis ou *functors* já criados no modelo. Por isso, após esta transformação, é necessário resolver essas referências, sendo isso levado a cabo na etapa 5, que tem como resultado modelos em MProlog.

Nas etapas 6 e 7 os modelos em MProlog são transformados num formato conveniente ao processamento para gerar prolog/texto - ou seja código fonte.

Na posse de prolog textual, podemos então gerar os testes abstractos na etapa 8, através de queries bem definidas a alguns predicados, e por fim na etapa 9 corremos esses testes contra o modelo EAPN para obtermos os oráculos. Para este modelo EAPN são também geradas algumas cláusulas que implementam a semântica operacional das EAPN, que permitem mutar o estado da EAPN perante determinado estímulo e daí concluir se esse estímulo é aceitável pela EAPN ou não. Não o sendo, conclui-se anotando o teste com um oráculo falso, caso contrário anotamos com oráculo verdadeiro.

Resta apenas referir que das etapas aqui apresentadas, as etapas 5, 6 e 7 foram previamente desenvolvidas em trabalhos do grupo de investigação nos quais esta dissertação se insere.

A descrição detalhada da solução será apresentada no capítulo 4.

1.4 Contribuições

Esta dissertaçãoX, como tem vindo a ser referido, tem como objectivo maior a geração de testes abstractos, sendo que se podem considerar como principais contribuições as seguintes abaixo enumeradas:

- Implementação da semântica operacional do SATEL, i.e., a derivação de testes abstractos baseada na semântica do SATEL;
- Composição do SATEL com a linguagem APN;
- Geração semi-automática de testes abstractos para sistemas modelados em EAPN;
- Fornecer uma semântica operacional para as EAPN

É possível considerar ainda como possíveis contribuições o facto de esta tese potenciar no futuro:

- A generalização da abordagem a outros formalismos de especificação de comportamento de sistemas;

- A integração com o model cheker AlPiNa da equipa ;
- A redução dos testes por critérios de equivalência.

1.5 Estrutura do documento

A presente dissertação encontra-se organizada da seguinte forma:

Primeiramente no capítulo 2 é introduzido o tema de teste de software, onde se apresentam os diversos tipos de teste existentes, dando maior ênfase ao teste baseado em modelos no fim do capítulo. No capítulo 3 introduzem-se noções sobre as APN e o SATEL, bem como a sua sintaxe e semântica. No capítulo 4 apresenta-se a solução encontrada para o problema, e os diferentes passos para se chegar a ela. Por fim apresenta-se o exercício prático, que será constantemente referido ao longo da dissertação, e as respectivas conclusões.

2

Estado da arte

Serve a presente secção para introduzir os diversos conceitos mais prementes e inerentes à temática abordada no âmago da dissertação, tais como o Teste de Software, seus métodos e variantes, ou particularização dos objectos de estudo. Apresentar-se-ão os vários tipos de teste de software existentes, culminando-se no teste baseado em modelos, que tem maior relevo para esta dissertação.

2.1 Teste de Software

O desenvolvimento de software tem sofrido grandes alterações ao longo do tempo relativamente às metodologias utilizadas. Em particular, com a pressão pela obtenção de software de qualidade, com o crescimento dos sistemas e, por consequência também da sua complexidade, tornou-se absolutamente prioritário criar métodos que não só acelerassem o desenvolvimento, como também garantissem que este era tão isento de falhas e incongruências em relação às especificações quanto possível.

No sentido de levar a cabo tal minimização foi introduzida uma sub-disciplina da Engenharia de Software - o Teste de Software -, que se pode considerar uma área aliada à Qualidade de Software, sendo que hoje encontramos diversos tipos de teste de software, uns mais convenientes que outros, consoante vários critérios como: os objectivos do teste, o alvo de teste, ou a escalabilidade dos sistemas.

Com o fim de contextualizar e definir "Teste de Software" podemos encontrar na literatura diversas tentativas conduzidas com o fim de suprir essa necessidade. De entre estas, encontram-se tanto visões intuitivas e pouco reveladoras como definições

formais com bases coerentes e bem fundadas [6, 13, 20], sendo que, em súmula, todas estas generalizam o mesmo tipo de definição; a de que "Teste de Software" consiste num processo (em ambiente controlado) de desenvolvimento de software, com o qual se consegue apurar a conformidade das execuções do software contra a sua própria especificação.

Não obstante as definições apresentadas, segundo a definição apresentada pela IEEE Software Engineering Body of Knowledge (SWEBOK) [10], "Teste de software" é uma actividade levada a cabo com o fim de avaliar e permitir o aperfeiçoamento da qualidade do produto, através da identificação de defeitos e problemas. Sob um ponto de vista mais detalhado, esta definição leva a considerar que o Teste de Software consiste na verificação dinâmica do comportamento de um programa, recorrendo a baterias de teste finitas convenientemente seleccionadas do domínio das suas execuções. Esta aferição é feita através do contraste com o comportamento previamente expectável, dados os casos de teste. [25]

Nesta definição, é de realçar que os termos "dinâmica", "finitas", "seleccionadas" e "expectável" não aparecem acidentalmente, mas exactamente porque todos eles são reveladores das características mais proeminentes do Teste de Software. Particularizando, atente-se no facto de o teste ser efectuado através de execuções reais ou simulações muito fidedignas do sistema, para as quais determinados *inputs* são dados. Isto reflecte o aspecto dinâmico do "Teste" por contraposição com algumas técnicas, ditas estáticas (e.g. reviews, análise estática como análise de referências nulas ou de alcançabilidade), em que a execução efectiva do software não é requerida. Mais se acrescenta que o teste como uma técnica dinâmica permite a avaliação imediata não só do código, como também de todas as partes integrantes do sistema, desde o compilador até ao sistema operativo. [25]

A finitude como característica do Teste de Software é inerente à escalabilidade dos sistemas. Sabe-se que os verdadeiros sistemas são ricos em termos de expressividade, permitindo a introdução de *inputs* cujo domínio é extenso e que, quando conjugados com as operações possíveis, podem levar a uma infinidade de estados, pelo que a execução de testes para todas as possibilidades de execução não é comportável, sendo necessário reduzir os casos de teste a um número finito e módico de modo a que a tarefa se torne possível no que diz respeito à complexidade temporal. No seguimento deste assunto (i.e. da dimensão do espaço de execuções do software), a selecção dos testes é também uma tarefa primordial para a execução dos mesmos e que deve obrigar a alguma ponderação. Na necessidade de seleccionar um conjunto de testes que possam majorar a cobertura de diferentes comportamentos obtidos pelas execuções do software, usualmente o perito de testes é forçado a analisar o conhecimento empírico que

possui sobre o sistema, de modo a seleccionar o mínimo número de testes. O óptimo desta minimização é atingido quando se consegue seleccionar um *input* por cada conjunto de *inputs* que levam o sistema a comportar-se de forma idêntica (assumpção de uniformidade). No que diz respeito ao estado da arte nesta matéria, existem algumas estratégias que podem acelerar o processo, tais como o teste por limitação dos intervalos de valores ou a redução a classes de equivalência, em que os testes são agrupados em classes pelo comportamento que despoletam no sistema, para posteriormente ser escolhido apenas um. Esta técnica é bem focada nesta dissertação, pois uma tarefa que integra o plano respectivo é exactamente esta redução a classes de equivalência, não manualmente mas de uma forma automática.

Por fim, o termo "expectável" surge ligado à fase final do processo de teste. Em rigor, após a execução de um teste, é necessário aferir os resultados; nomeadamente é preciso concluir se o sistema apresenta uma falha ou não, de acordo com o teste realizado. A este processo dá-se a designação de "problema do oráculo", sendo que oráculo é o processo de comparação dos *output* do SUT (*System Under Test*/Sistema sob teste) com o *output* que o oráculo estabelece como correcto para um determinado *input*, ou seja é um método de despistagem da falha do teste. Neste sentido, torna-se óbvio que os oráculos estão sempre separados do SUT. Para finalizar a definição de oráculo resta referir que estes podem ser de diferentes naturezas, e nesse sentido, incluir especificações e documentação [21], ou uma verificação humana que ateste o correcto funcionamento [15]. Podem basear-se numa heurística (oráculo de heurística) que fornece resultados sobre o conjunto de teste, ou em critérios de consistência (oráculo de consistência) que comparam os resultados da execução de um teste com outro, ou ainda um oráculo baseado em modelos que utiliza o mesmo modelo para gerar e verificar o comportamento do SUT [23]. Pode assim concluir-se que "Teste de Software" é uma tarefa dinâmica (uma vez que os SUT são realmente executados). Visa verificar a existência de falhas no sistema e identificar a respectiva causa, por meio da utilização de um subconjunto finito de testes, cuja selecção é feita tendo em vista a obtenção de uma amostra representativa (no que diz respeito aos diferentes comportamentos do sistema), e da confrontação da execução do teste com um oráculo. Mais se conclui que "Teste de Software" é uma actividade complementar de outras formas de verificação estática e é, também, um processo distinto do chamado *debugging* (correcção de código), levado a cabo para correcção da causa das falhas identificadas.

2.2 Tipos de Teste

Existem distintas formas de se efectuarem testes sobre um SUT, sendo que em particular há testes de abstracção mais baixa e de abstracção mais alta, que se podem classificar, respectivamente, em testes de caixa-branca (*white-box testing*) e testes de caixa-preta (*black-box testing*).

Em rigor, podemos observar na figura 2.1 um eixo tridimensional que sugere uma forma de classificação dos diferentes tipos de teste em função da dimensão do SUT (desde pequenas unidades de código (e.g. uma classe) até todo o sistema), das características que se pretende testar (e.g. Funcionalidade, Robustez, Performance, Usabilidade,...), e do tipo de informação utilizada para desenhar os testes (e.g. *black-box* em que se desconhece a estrutura interna do SUT, e *white-box* em que é utilizada a implementação do software para desenhar os testes).

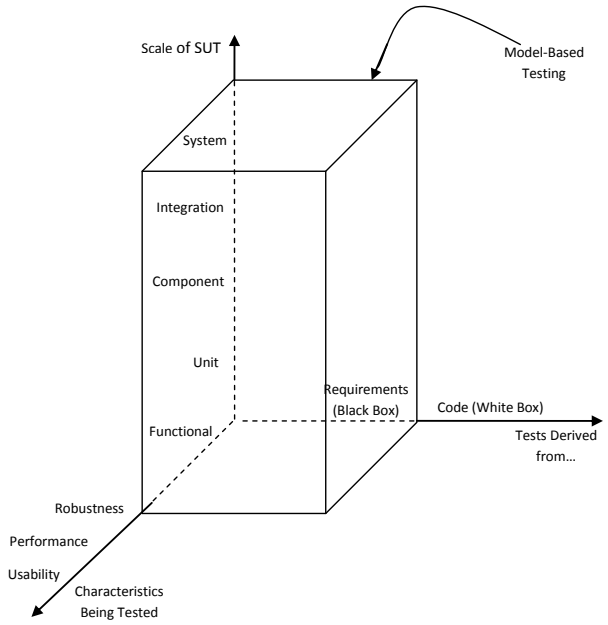

Figura 2.1: Os três eixos de caracterização de teste. (Adaptado de [25])

De um modo mais detalhado, no eixo da dimensão do SUT, pode classificar-se o teste ao nível da unidade (teste de simples classes ou procedimentos), ao nível dos componentes (realizam-se testes a cada subsistema isoladamente), ao nível da integração (para se garantir a correcta colaboração entre componentes), e ao nível de todo o sistema.

No que diz respeito ao eixo das características do SUT a serem testadas, pode-se classificar o teste como funcional/comportamental, que visa despistar erros de funcionalidade, (i.e., verifica se as funções estão correctamente implementadas) como por

exemplo os erros de *output*. Nesta gama, existem ainda os testes de robustez que visam procurar falhas de execução em condições inválidas em geral como *inputs* inválidos ou indisponibilidade de rede; os testes de performance/stress que avaliam o desempenho (e.g. tempo de resposta e funcionamento) do SUT em diferentes contextos (e.g. situações de sobrecarga de sistema), e ainda os de usabilidade que visam testar a existência de problemas de interface com o utilizador. A estes poder-se-iam acrescer outros tipos de teste como os de recuperação, que forçam falhas no sistema visando observar a resiliência do mesmo, ou os testes de segurança, em que o *tester* poderá assumir a posição de *hacker*, procurando identificar focos (causas) de possíveis falhas de segurança do SUT e respectivos dados.

Em relação ao eixo do tipo de informação utilizada, os testes de caixa-branca são efectuados objectivamente no código e geralmente são feitos pelo próprio programador (os testes de unidade são um exemplo deste tipo de testes). Sob outro ponto de vista, os testes de caixa-preta são efectuados de forma funcional, onde não existe acesso ao código do sistema e em que este é visto como uma caixa opaca, onde apenas entram dados e saem resultados.

Mais se acrescenta que, conforme [19], os testes de caixa-preta mais utilizados são os testes funcionais, os de robustez e os de aceitação que são similares aos primeiros, com a diferença de serem efectuados pelos clientes. Por fim, existem também os testes mistos, que tanto são de caixa-branca como de caixa-preta, os testes de regressão e os de integração, sendo que os primeiros são, ou devem ser, efectuados a cada alteração considerável do sistema. Concluindo, resta referir que o *Model Based Testing*, tal como se pode inferir a partir do eixo de classificação, é aplicável a qualquer um dos níveis de dimensão do SUT, e em geral é utilizado para gerar testes funcionais, podendo ser utilizado em alguns testes de robustez como o da inserção de *inputs* inválidos, sendo que ao nível dos testes de performance não existe ainda grande aplicação, tal como se afirma em [25].

Em relação ao eixo restante, como no "Teste Baseado em Modelos"os testes são derivados da especificação e respectivos requisitos, conclui-se que se deve enquadrá-los numa filosofia de caixa preta já apresentada, uma vez que se parte dos requisitos e se procura inferir sobre a sua satisfação.

2.3 Teste baseado em modelos

O Teste baseado em Modelos (*Model-based testing, MBT*) surge naturalmente na linha de investigação do "Teste de Software"e, em particular, surge da necessidade de automatizar a actividade em questão de forma a tornar mais efectiva a geração de testes

abstractos a partir de modelos que abstraem o comportamento do SUT, e com menores custos em relação àqueles que se teriam caso o SUT fosse testado directamente. A acrescentar há ainda o facto de que quando são modelos especificados de forma operacional, as abordagens MBT fornecem um oráculo para os testes gerados [25].

Detalhando o MBT de um modo mais formal, deve referir-se que este resulta do aproveitamento das abordagens de desenvolvimento orientadas aos modelos para desenvolver os elementos necessários à execução de testes de software. Em particular, neste tipo de abordagem deverá existir um modelo que compreenda os detalhes de diversos aspectos do SUT do qual é derivado um modelo de testes. Este modelo de teste deve na sua essência descrever alguma informação necessária à execução dos testes (e.g. casos de teste, ambientes de execução dos mesmos, etc).

Já o modelo do SUT, no contexto do MBT, deve descrever em particular aspectos funcionais já que está em causa, tal como se referiu previamente, o desenho de testes de caixa-preta.

Sendo uma abordagem que se firma sobre modelos é importante conhecer o nível de abstracção que os caracteriza. Em rigor, o modelo do SUT deve modular a componente funcional do sistema de uma forma suficientemente abstracta, ou seja, deve modular acima de tudo as saídas do sistema em função das respectivas entradas. Dado que o conjunto de casos de teste abstratos (*abstract test suite*) é gerado por derivação do modelo do SUT, conclui-se que ambos os modelos (do SUT e de teste) se encontram no mesmo nível de abstracção (figura 2.3).

Este nível de abstracção é diferente (mais alto) daquele em que ocorrem as execuções do SUT, e por esse mesmo motivo o conjunto de casos de testes abstractos não pode ser executado directamente sobre o SUT, tornando-se necessário fazer uma derivação de um conjunto de casos de testes executáveis a partir do conjunto de casos de testes abstractos que possam correr e interagir directamente sobre o SUT. Isto é feito, obviamente, por meio de mapeamento entre os testes abstractos e os testes executáveis, recorrendo a dados presentes nos modelos de teste que servem para esse propósito.

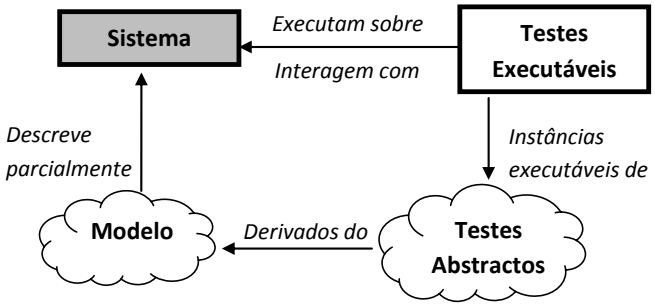

Figura 2.2: Abordagem de teste baseada em modelos

Em relação ao Teste baseado em Modelos, de acordo com [25], sabe-se que esta nomenclatura é usada para diferentes abordagens de teste com modelos, que diferem essencialmente no tipo de modelo que é utilizado, e por conseguinte não só na forma como é feito o teste, mas também nos proveitos que se conseguem obter.

Neste sentido aparecem na literatura e na indústria a geração de testes a partir de modelos de domínio, de modelos de ambiente, de modelos funcionais (comportamento do sistema) e de modelos descritores da estrutura do SUT.

Em relação à geração de *inputs* a partir de modelos de domínio é de salientar o facto de os modelos conterem informação sobre o domínio dos valores de *input*; na verdade, nesta abordagem a geração consiste na selecção e combinação dos valores dos domínios dos *inputs*, utilizando algoritmos mais ou menos sofisticados, consoante a implementação requerida. Este tipo de abordagem, embora sendo importante, peca por não suportar o conceito de oráculo que permite verificar se o teste passa ou não, pelo que se depreende que esta abordagem não soluciona todo o problema da produção de testes.

Por outro lado tem-se a geração de casos de teste a partir de modelos de ambiente operacional, como por exemplo os modelos estatísticos sobre a frequência da solicitação de operações do SUT. Nesta abordagem geram-se sequências de chamadas a operações interpretáveis directamente pelo SUT, todavia à semelhança do que acontece com a anterior, as sequências não discriminam informação sobre os *outputs* expectáveis, pelo que o problema anterior subsiste nesta abordagem.

Outro tipo de abordagem MBT consiste na transformação de descrições abstractamente elevadas de casos de teste, em *scripts* de teste executáveis (baixo nível de abstracção por natureza), sendo que as descrições podem ser feitas com linguagens de modelação/especificação como UML, recorrendo aos diagramas inerentes (e.g. sequência). Numa abordagem desta natureza, os modelos são, não só conjuntos de informação sobre a estrutura interna do sistema, como também a própria API do SUT e as regras de transformação das chamadas de alto-nível de abstracção em scripts de teste executáveis.

Por fim, tem-se uma abordagem que toma partido dos modelos funcionais (descritores do comportamento do sistema) para gerar casos de teste com oráculos. *Esta abordagem, ao contrário do que ocorre com as supracitadas, possibilita a geração de oráculos que permitem aferir sobre o sucesso do teste, e em particular é a abordagem com maior relevância nesta dissertação, uma vez que o trabalho proposto segue a mesma filosofia.* O objectivo principal é gerar casos de teste que sejam executáveis e que incluam informação dos oráculos, tal como os valores de *output* esperados, para que se possa a posteriori verificar se estes coincidem com os valores que o SUT efectivamente devolve, aferindo

assim o sucesso/insucesso do teste.

Em relação às restantes estratégias já salientadas, esta requer o esforço adicional da geração dos oráculos e da verificação no final do processo. Neste caso, e tendo em conta esta observação, é necessário que o conteúdo do modelo descreva o comportamento do SUT (e.g. descrição dos *outputs* e *inputs* do sistema e da relação entre ambos,...), para que seja possível efectuar todo o percurso de resolução do problema da geração de testes, que tem início na selecção dos valores de *input*, e finda com a análise de coincidência entre oráculos e *outputs* do SUT, visando aferir o resultado do teste.

2.4 Teste: os diferentes processos

Glenford J. Myers (autor de "The Art of Software Testing") afirmou em [20], que:

> Teste é o processo de executar um programa com o intuito de encontrar erros.

Desta proposição percepciona-se o teste como um processo, pelo que se torna premente entender o ciclo de vida do mesmo, bem como a forma como é conduzido.

De [25] discorre que o Teste de Software compreende três fases principais. São elas: o desenho dos casos de teste (que deve ser feito a partir dos requisitos do sistema, tendo em conta os objectivos de nível mais elevado definidos para cada teste), a execução dos testes e análise de resultados (i.e., a execução dos testes sobre o SUT e a análise de resultados de forma a poder concluir-se sobre o sucesso/falha do teste), e por fim a verificação da cobertura dos requisitos por parte dos testes (como forma de capturar o nível de qualidade oferecido pelo produto (software de teste) que é reflectido pelo grau de cobertura dos requisitos por parte do produto).

Ainda de acordo com [25], existem diversos processos de teste; desde o simples processo totalmente manual até aos mais automatizados, como é o caso do processo de teste orientado por modelos. Sendo este último o de maior relevância para esta dissertação, faça-se apenas uma pequena súmula dos processos menos automatizados.

O processo com maior antiguidade (contudo ainda hoje amplamente utilizado) é aquele cuja execução é efectuada através de meios inteiramente manuais. Neste tipo de processo, tal como se pode observar na figura 2.3, o plano de teste deve ser constituído por uma descrição de alto nível dos objectivos do teste e a elaboração dos testes é feita por via manual, alicerçando-se nos requisitos expressos de forma informal, sendo que desta etapa deve resultar um documento sucinto que contenha a descrição (de alto nível) dos casos de teste. A execução dos testes é também feita de forma manual e pressupõe, à partida, que o engenheiro de teste possui o conhecimento necessário sobre o

SUT que lhe permite não só executar os testes, como também mapear directamente as descrições de alto nível em instruções de baixo nível, executáveis directamente sobre o mesmo. O processo de análise é, por definição, também ele feito de forma manual. O *tester* é responsável pela execução dos testes sobre o SUT, por efectuar a comparação dos *outputs* esperados com os *outputs* obtidos e pelo registo dos respectivos resultados.

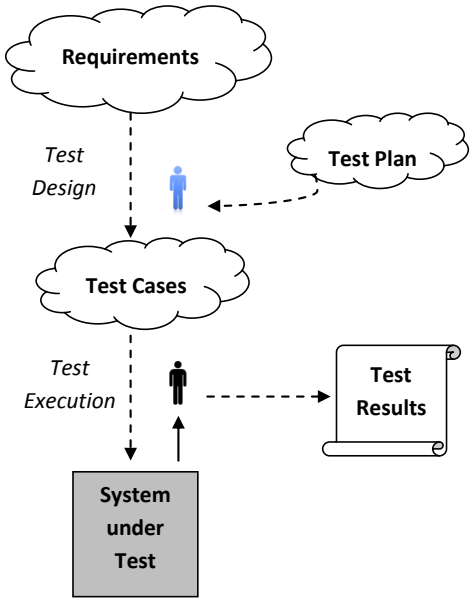

Figura 2.3: Teste segundo o processo manual

Segundo [25], este tipo de processo apresenta algumas desvantagens não só porque para o levar a cabo é, claramente, necessário um dispêndio acrescido de tempo e de recursos em geral, como também por estar dependente do factor humano, das suas falhas e idiossincrasias. Neste sentido, deve-se sublinhar também que a cada alteração periódica (com período devidamente definido), novos testes devem ser efectuados ao SUT, e posto que a tarefa é feita manualmente, as etapas do processo têm que ser repetidas, conduzindo a um custo de teste elevado.

Estas desvantagens e, em particular, a questão das reiterações manuais sucessivas remetem para a necessidade de automatização dos processos. É com este intuito que surgem processos similares ao exposto, mas que acrescentam algum tipo de processamento automático nalguma fase do processo. Posto isto, são de salientar o processo de Teste *Capture/Replay* (2.4) e o processo de Teste baseado em *scripts*. O primeiro resulta da tentativa de colmatar o problema das reexecuções sucessivas dos testes. Para esse efeito é introduzida uma ferramenta *Capture/Replay* responsável pela interacção com o SUT. Na prática a ferramenta é como que um observador activo com a função

de *cache*, que regista cada *input* dado como teste e os respectivos *outputs*.

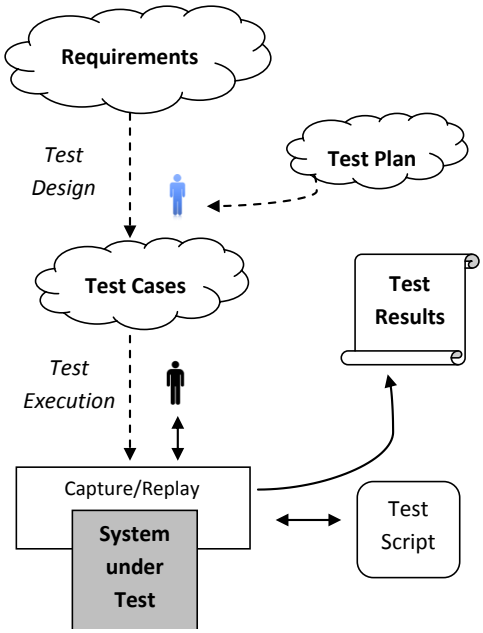

Figura 2.4: Processo de Teste Capture/Replay

Na realidade, a supressão do problema é suficientemente efectiva porque em testes futuros, a ferramenta socorre-se dos *inputs* em cache para efectuar os testes, e dos respectivos *outputs* para fazer a validação dos *outputs* dos novos testes. Este tipo de processo resolve a questão da repetição, no entanto, encerra novos inconvenientes em virtude da volatilidade dos testes. Em particular, repare-se que existe a possibilidade de diminutas modificações no SUT (i.e. o sistema pode ser alterado), que podem invalidar a execução de alguns testes anteriormente válidos. Nestes casos é necessário modificar também os testes de forma a que possam ser executados, o que pode implicar uma sobrecarga de actualizações não desejáveis, pelo que foi exposto para o primeiro processo. Isto pode significar a não viabilidade do processo pela simples razão de os testes não serem suficientemente abstractos para poderem ser executados entre diferentes estágios de desenvolvimento do SUT.

Na sequência do que foi previamente expresso sobrevêm processos que tornam automáticas, de algum modo, outras etapas da geração de testes. Em particular, o já enunciado processo de geração de testes com base em *scripts* de testes surge da tentativa de tornar o processo de execução dos testes inteiramente automático, removendo a bilateralidade entre o componente humano e o motor de testes presente na figura 2.4, isto é, removendo a necessidade de sucessivas acções de manutenção dos testes. Este tipo de processo recorre-se da programação de *scripts* de teste, que correm diversos casos de teste e que, em geral, são responsáveis por diversas operações do SUT (definição

do contexto de execução, introdução de *inputs*, registo e análise de *outputs*,...). Para o processo funcionar é ainda necessário que o SUT ofereça mecanismos de controlo e observação do sistema (potencialmente por meio de uma API) de forma a se poder verificar se o sistema se comporta segundo o que é expectável em cada um dos estados. Da conjunção de toda esta caracterização remanesce o problema da manutenção, já que os *scripts* devem estar de acordo, quer com os requisitos, quer com os detalhes de implementação, por via de utilização de uma API.

Como previamente referido, a lógica para a resolução dos problemas enunciados centra-se na elevação da abstracção dos casos de teste e na automatização dos seus diversos estádios. É neste sentido, de acordo com [25], que foram desenvolvidos outros tipos de processos, tais como o teste conduzido pelos dados, por tabelas, por acções e/ou por palavras-chave, que por essência procuram reduzir a questão da manutenção de cariz eminentemente humano. Mais se acrescenta que para estes, o objectivo é abstrair, tanto quanto possível, os casos de teste até um nível em que possam ainda ser interpretados por uma ferramenta de execução. Na prática, e não pretendendo a exaustão do assunto (já que não é peremptório para esta dissertação), o que é feito nos processos conduzidos por palavras/acções/... é definir alguns símbolos (termos, palavras-chave) que correspondam a fragmentos de scripts e que possam ser convertidos por meio automático (dito de adaptador), em testes executáveis. Pode concluir-se, portanto, que este último tipo de processo situa os casos de teste num patamar mais alto de elevação, reduzindo a maior parte dos problemas de manutenção. Não obstante, mantém o problema da manualidade da geração de dados para teste, dos oráculos e da verificação de cobertura dos requisitos de teste.

É na sequência da tentativa de resolução dos problemas enunciados que surge o processo de Teste baseado em Modelos, sendo que, em particular, com este é esperada a automatização do desenho de casos de teste funcionais (inclui predição dos *outputs*) de forma a que se veja reduzido o custo de desenho e que se produzam conjuntos de teste que cubram efectivamente o modelo. Pretende-se ainda com este processo a redução de custos de manutenção e a geração automática da matriz de confrontação entre requisitos e casos de teste, dita de matriz de rastreabilidade.

2.5 O processo de teste baseado em modelos

O processo de Teste Baseado em Modelos introduzido anteriormente, supre a criação manual dos casos de teste através da introdução de uma etapa em que o *test designer* cria um modelo abstracto do SUT. Obtendo-se um modelo é possível não só extrair casos de teste a partir do modelo de modo automático, como também fazer essa ex-

tracção de diversas formas consoante a definição de diferentes critérios de selecção, contribuindo para a redução efectiva do tempo de criação de testes [25].

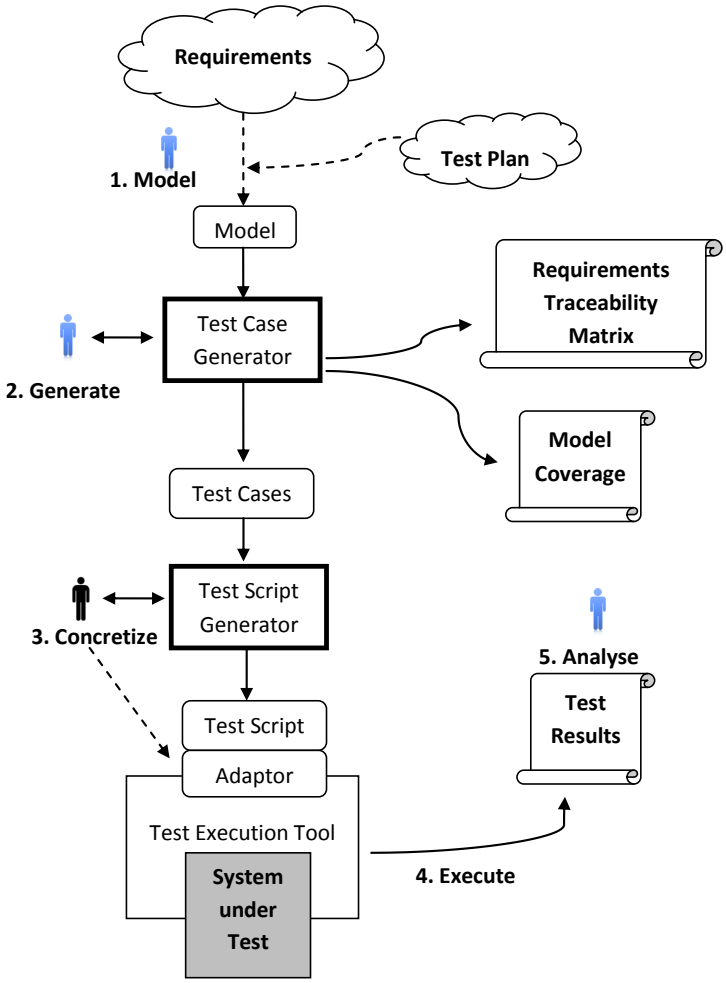

Figura 2.5: Processo de Teste baseado em Modelos

Atentando-se na figura 2.5, ressaltam cinco etapas principais: duas que distinguem este processo dos restantes (criação de um modelo do SUT e geração de testes abstractos a partir do modelo), a transformação dos testes abstractos em testes executáveis, e por fim as que são comuns a qualquer processo já enunciado (execução dos testes e obtenção e classificação de *outputs*, e análise de resultados).

Em abono da boa compreensão de cada uma das etapas supracitadas, é fundamental perceber no que consiste cada uma delas. Procedendo a uma análise detalhada fica claro que na primeira etapa é desenvolvido um modelo abstracto do sistema, assim intitulado por se se pretender conciso, abstraído dos detalhes e consequentemente de tamanho reduzido em relação ao tamanho do sistema; deve ainda permitir a identificação directa das relações entre o modelo e os requisitos. Por outro lado é importante que

o modelo seja implementado em ferramentas que permitam a sua validação/verificação automática através de mecanismos embutidos (verificação de conformidade entre modelo e metamodelo), sendo que em particular, no contexto da presente dissertação, a ferramenta em desenvolvimento recorre-se da plataforma de modelação do Eclipse (EMF-*Eclipse Modelling Framework*[8]) que possui os mecanismos para esse efeito.

Na segunda etapa - a geração de testes abstractos - o objectivo primordial é a obtenção de um conjunto de testes abstractos através da definição de alguns critérios de selecção que permitam guiar a escolha dos mesmos. Em geral, existe uma infinidade de testes passíveis de ser seleccionados e, não sendo possível a geração de todos, têm de ser indicadas explicitamente quais as partes do modelo que devem ser testadas com o fim de restringir a geração. Os critérios podem ser expressos directamente no *software* de teste em questão através de opções disponibilizadas ou, por outro lado, pode ser especificado através de uma linguagem de especificação de testes. Esta última opção é a que mais se coaduna com esta dissertação, já que na ferramenta em desenvolvimento os testes são especificados através de uma linguagem de definição de intenções de teste (SATEL), que será abordada posteriormente em capítulo próprio.

A terceira etapa tem como objectivo a transformação dos testes abstractos em testes executáveis directamente sobre o SUT. Este objectivo é concretizável por exemplo através da utilização de ferramentas de transformação que usam templates e/ou mapas de relações com o fim de traduzir cada teste abstracto num teste/script executável. A quarta etapa consiste simplesmente na execução efectiva dos testes sobre o SUT. Na realidade esta etapa pode estar fundida na segunda etapa - geração de testes abstractos - no caso em que todo o processo é feito de forma imediata como um todo. Nesta situação, assim que o teste é produzido, a ferramenta deve executá-lo e recolher a informação de resultados. Dito de outro modo, desta forma as etapas são sucessivamente dependentes e são obrigatoriamente realizadas nos mesmos instantes, em oposição ao teste *off-line*, em que as etapas, embora dependentes entre si, permitem -se à execução em instantes temporais distintos.

Por fim a quinta etapa confina-se à análise dos resultados obtidos com as execuções e à correcção das causas das respectivas falhas. Sendo um processo baseado em MBT, as causas podem situar-se quer ao nível do modelo, quer ao nível do adaptador (*software* responsável por mapear os testes abstractos em executáveis), como também ao nível dos documentos de especificação de requisitos.

Das etapas acima, apenas a primeira e a segunda têm relevância na presente dissertação, já que apenas há interesse na geração de testes abstractos e do oráculo a partir de um modelo, não se tendo o objectivo de concretizar os testes abstractos em testes executáveis sobre o SUT.

2.6 Ferramentas de teste baseadas em modelos

As ferramentas MBT disponíveis procuram acima de tudo integrar conjuntamente as etapas de análise de requisitos e de teste, que correspondem a duas etapas distintas no processo de MBT. Sob um ponto de vista positivista, seria óptimo automatizar todo o processo, para que os *inputs* das ferramentas fossem os requisitos e os *outputs* fossem os casos de teste incorporando os *outputs* espectáveis do SUT. Não obstante, existem diversas ferramentas de MBT, quer comerciais quer académicas, que se focam na sua maioria ao nível da transformação dos testes abstractos em *scripts* de testes executáveis. Estas permitem ainda controlar, dentro de uma certa granularidade, a selecção e cobertura dos testes. A tabela 2.1 exibe de modo sucinto algumas dessas ferramentas.

Como se pode observar, na tabela estão representados os quatro tipos de abordagens MBT apresentados anteriormente no sub-capítulo que lhe foi destinado e, em particular, foram evidenciadas através de sombreado aquelas que se baseiam em modelos comportamentais do SUT para gerar casos de teste e respectivos oráculos, tal como a ferramenta proposta nos trabalhos da presente dissertação. A título exemplificativo denote-se também as ferramentas que geram dados de *input* a partir de modelos de domínio (e.g. AETG Web Service...), as ferramentas que geram casos de teste a partir de modelos de ambiente do SUT (e.g. Ma TeLo) e, por fim, as ferramentas que geram *scripts* de teste a partir de testes abstractos e que, em particular, partem de notações abstractas, ou similarmente de alto nível (e.g. TAU Tester).

Ainda da tabela, repare-se nos diferentes tipos de notações de modelação utilizados pelas diversas ferramentas.

Para além das ferramentas apontadas nesta tabela podem ser ainda enunciadas outras com grande projecção no meio científico, como é o caso da ferramenta Spec Explorer[1] desenvolvida pelas equipas de investigação da Microsoft®.

Esta é já uma ferramenta amadurecida de MBT que permite testar sistemas reactivos de software orientado a objectos, sendo que através da notação de modelação Spec#, o Spec Explorer [26] cobre aspectos como a criação de objectos dinâmicos, não determinismo e comportamento reactivo, análise de modelos, teste *online* e *offline* e execução automática de testes.

O Spec# é uma ferramenta que descende de uma versão light do AsmL[2], com alguns pontos de similaridade com o ESC/Java para Java, mas que por seu turno assenta sobre código C#. Com esta notação desenvolve-se o código do sistema anotando-o com pré-condições, pós-condições, invariantes, invariantes de ciclo, verificação de desrefe-

[1] http://research.microsoft.com/en-us/projects/specexplorer
[2] http://research.microsoft.com/en-us/projects/asml

Ferramenta	Organismo	Inputs da ferramenta	Outputs da ferramenta
AETG Web Service	Telcordia Technologies Applied Research	Parâmetros de Input na forma tabular	Casos de Teste
Abstract State Machine Language (AsmL)	Microsoft	XML e palavras	Geração de testes baseada na cobertura total das transições de um autómato finito.
Conformiq Test Generator	Conformiq Software, Limited	Diagramas de Estado UML	Casos de teste no formato TTCN, incluindo os resultados expectáveis.
Direct-To-Test(DTT)	Software Prototype Technologies	Modelos em linguagem própria, tabelas causa/efeito	Casos de teste incluindo outputs esperados e scripts de testes executáveis
GOTCHA-TCBeans	IBM Research Laboratory in Haifa	Modelos em linguagem específica	Casos de teste incluindo outputs esperados e plataforma de tradução de testes
Ma TeLo	All4Tec	Editor de Modelos de Utilização (Estatísticos) através de Cadeias Markovianas	Gera casos de teste a partir dos modelos de utilização de um SUT
MulSaw	Massachusetts Institute of Technology	Modelos na linguagem de modelação Alloy ou em JML, incluindo pre e pos condições	Testes baseados em critérios de cobertura
Reactis	Reactive Systems, Incorporated	Modelos em Simulink(MatLab) e Linguagem de Modelação Stateflow	Simulador de Modelos, conjunto de testes, incluindo resultados expectáveis.
SDL And MSC based Test case Generation (SAMSTAG)	University of Fribourg	SDL system specifications, MSC test purposes	Casos de teste no formato TTCN.
SmartTest	Smartware Technologies	Diagramas de classe e UML	Geração de Testes através de técnicas/algoritmos de pairwise
SpecTest	George Mason University	Modelos em SCR ou UML	Testes baseados em critérios de cobertura
TAU Tester	Telelogic	Modelos TTCN-3	Ambiente de execução para testes TTCN-3
Test Generation with Verification (TGV)	IRISA and VERIMAG Laboratories	Especificação de modelos LOTOS, SDL, ou IF	Casos de teste no formato TTCN, incluindo os resultados expectáveis.
Test Vector Generation System (TVEC)	TVEC Technologies	Modelo comportamental em linguagem proprietária ou em modelos SCR	Casos de teste abstractos e programa de teste executável
The Object-oriented Software Testing Environment (TOSTER)	Warsaw University of Technology	Diagrama de estados UML	Gera e executa os casos de teste e gera os outputs expectáveis
Test Cover	TestCover	Modelo de domínio dos dados de input	Geração de casos de teste recorrendo a técninas de pairwise
TorX	University of Twente	Modelos LOTOS, PROMELA, ou SDL.	Geração de casos de teste
ZigmaTEST	ATS	Modelos (Máquina de estados finita)	Sequência de testes que cobrem as transições da máquina de estados

Tabela 2.1: Ferramentas de Model-Based Testing (adaptado de [12,25])

renciação nula,... que permitem a verificação estática do sistema.

O Spec Explorer surgiu tentando colmatar algumas falácias do MBT, como a falta de soluções adequadas para lidar com a explosão de estados, utilização de linguagens de especificação em vez de linguagens de programação correntes, falta de suporte para modelação orientada a cenários (em particular para a composição de cenários com máquinas de estados) e por fim para lidar, quer com a falta de integração com IDEs em

voga, quer com motores de execução já existentes.

Desta forma, esta ferramenta aceita programas em C# como *input*, faz pesquisa sobre o modelo baseada em CIL (código da máquina CLR da plataforma .Net) e recorre-se de *scripts* em linguagem Cord[3], que é uma linguagem de *scripting* para configurar a exploração e teste do modelo, para descrever a composição do modelo e cenários. Estes *scripts* são responsáveis por mapear o modelo no SUT com declarações das acções correspondentes aos métodos que têm interesse na implementação do SUT; permitem ajustar alguns valores como, por exemplo, os limites de exploração do modelo; adicionam máquinas de estado ao modelo, que podem ser combinadas e exploradas para determinar largura e profundidade da cobertura dos testes em relação ao SUT e ainda fornece meios para gerar testes a partir do modelo.

Neste momento o projecto Spec Explorer continua ainda em linha de investigação em duas vertentes, uma de teste de caixa-branca e outra de MBT.

No que concerne à presente dissertação, a ferramenta a desenvolver recorrerá a modelos comportamentais do SUT especificados num formalismo algébrico (Petri nets algébricas) e, diferentemente do Spec Explorer, acenta numa abordagem conduzida por modelos, para guiar a geração dos testes. Além disto é uma ferramenta na qual os testes são caminhos de execução do SUT, com os quais se pode estimular o sistema e observar os resultados.

2.7 Vantagens do Teste baseado em modelos

Tendo sido introduzidas as várias abordagens MBT, os diversos detalhes do seu processo e ainda algumas ferramentas que lhe servem de base, não será menos importante perceber que tipo de benefícios decorre da utilização deste processo. Apesar de ao longo dos últimos capítulos terem vindo a ser introduzidas implicitamente algumas das vantagens mais notórias, de modo sucinto pode afirmar-se, segundo [25], que o MBT promove por um lado a diminuição do tempo e custo de teste e, por outro, permite o aumento da qualidade do *Software*.

De outro modo possibilita não só a exploração mais precoce de ambiguidades e falhas decorrentes das fases de especificação e modelação, como também a geração automática e isenta de repetições de um número elevado de testes práticos como pode ocorrer em processos de maior manualidade. Outro tipo de mais-valias decorre do facto do MBT facilitar a actualização do conjunto de testes aquando de passíveis alterações dos requisitos (já que isso deve significar, normalmente, apenas alterações no modelo) e do facto de tornar mais fácil a compreensão do nível de cobertura, do mo-

[3]http://msdn.microsoft.com/en-us/library/ee620399.aspx em 16/12/12

delo e dos requisitos, oferecida pelo conjunto de testes.

Ainda de [25], é possível extrair um exemplo que ilustra de forma indubitável as diferenças entre o MBT e os restantes processos de teste. A tabela 2.2 mostra o número de horas necessárias para efectuar o teste de cada versão de uma determinada aplicação para cada um dos diferentes métodos de teste e a figura 3.6 mostra-nos o número acumulado de horas exigidas para fazer face ao teste das dez versões da aplicação.

Activity	Test(No.)	Manual	Replay	Script	Keywd	MBT
Initial Manual Test Design		50	50	50	50	0
Initial Modeling						30
Initial Test Generation						15
Initial Adapter Coding				50	15	15
Initial Test Execution		30	30	2	2	2
Total for Version 1	300	80	80	102	67	62
Total for Version 2	330	38	23	20	15	14
Total for Version 3	363	42	25	21	15	14
Total for Version 4	399	46	28	23	16	14
Total for Version 5	439	51	31	25	16	14
Total for Version 6	483	56	34	28	17	14
Total for Version 7	531	61	37	30	18	14
Total for Version 8	585	67	41	33	18	14
Total for Version 9	643	74	45	36	19	14
Total for Version 10	707	81	49	40	20	14
Total Hours		596	392	358	220	188

Tabela 2.2: Número de horas dispendidas por pessoa no teste de cada versão (de [25])

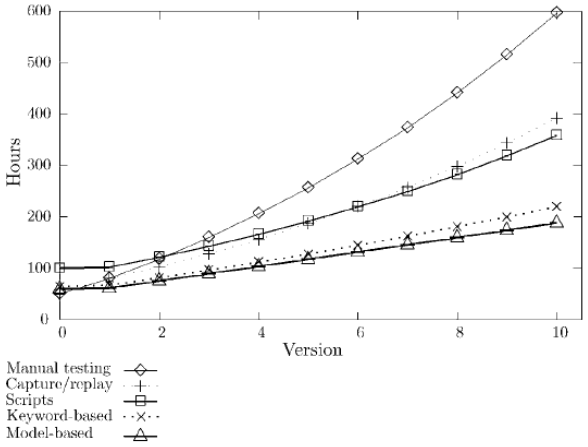

Figura 2.6: Total de horas de teste por cada processo de teste (de [25])

Tanto da tabela como do gráfico se depreende uma minoração dos consumos de horas perpetuada pelo processo MBT ao longo das sucessivas versões. Em particular, pode concluir-se algo mais do que isto. Ao longo dos diferentes capítulos, a necessidade de abstracção dos testes foi bem perceptível, e isso é extraível do gráfico. Repare-se que os testes manuais são os que apresentam custos iniciais mais baixos, todavia,

com o aumento da complexidade das versões os custos crescem quase que exponencialmente; já os testes efectuados a partir de *scripts* de forma automática apresentam um custo inicial maior, mas acabam por não ter um crescimento tão acentuado ao longo das versões, como acontece com o teste manual.

Por fim, o MBT pode garantir custos baixos inicialmente e, independentemente disso, assegura a sua minimização ao longo das restantes versões. Isto justifica-se com o desenho de testes com alto nível de abstracção, que facilitam todo o processo de manutenção, desenho e automatização, pelos motivos apresentados nos capítulos anteriores.

Model-Based Testing é a técnica utilizada nesta dissertação não só porque é a que se apresenta na teoria e na prática como o método mais eficiente, mas também porque a própria linguagem de especificação de testes (SATEL) é, ela própria, uma linguagem desenhada para servir os propósitos das abordagens orientadas a modelos.

2.8 Abordagens de geração de casos de teste a partir de especificações algébricas

Tendo em conta que esta dissertação se centra numa abordagem baseada em modelos especificados através de estruturas algébricas, torna-se premente mencionar quais as técnicas que se podem identificar para levar a cabo a geração de casos de teste nesse contexto.

Segundo [11] a reescrita de termos é uma dessas técnicas e consiste fundamentalmente em utilizar os axiomas do tipo de dados abstractos para transformar o lado esquerdo de uma equação no lado direito. Neste tipo de técnica torna-se difícil gerar casos de teste quando a aplicação dos axiomas depende de condições.

Uma outra técnica descrita em [9, 14] é a configuração manual. Esta peca por ser morosa, errónea (uma vez que contempla o factor humano) e pouco eficiente no que diz respeito à grande combinatória de casos de teste gerados.

Pode-se mencionar ainda a técnica de substituição de variáveis referenciada em [8,9,16,27], na qual são gerados valores aleatórios para as variáveis livres dos axiomas. Devido ao factor aleatório nem sempre se consegue encontrar valores que satisfaçam as restrições impostas aos axiomas; por outro lado uma geração exaustiva em profundidade pode culminar também no insucesso da pesquisa de valores apropriados por motivos de complexidade temporal, tal como se verá no *términus* desta dissertação.

De uma forma geral pode apontar-se falhas a qualquer uma das técnicas, no entanto segundo [2] todas estas técnicas falham por não contemplarem uma forma de geração de testes para tipos de dados abstractos genéricos (GADTS). Problema para o

qual em [2] é sugerida uma forma de geração fazendo uso do analisador Alloy Analyser [1] para encontrar modelos que satisfaçam condições que correspondem aos casos axiomáticos a testar.

Esta abordagem começa por diferir da apresentada nesta dissertação, no ponto em que a geração de testes apenas é baseada nas especificação de tipos de dados algébricos, não contemplando por isso a geração de testes mais sofisticados baseados em especificações comportamentais, tais como redes de Petri algébricas. Além disso pretende-se não só explicitar um modelo do SUT, mas também o comportamento do SUT que se pretende observar. No mesmo sentido, pretende-se guiar a geração dos testes através da especificação de um caminho de execução pretendido, ao contrário da abordagem em [2], em que os testes são apenas obtidos a partir da definição axiomática de um tipo de dados.

3

Trabalho relacionado

No presente capítulo serão introduzidos alguns formalismos tais como os tipos de dados algébricos e as redes de Petri algébricas que a eles recorrem. Introduzir-se-á também posteriormente a linguagem SATEL tal como foi concebida pelo autor e a respectiva semântica.

3.1 APN - Redes de Petri algébricas

Redes de Petri Algébricas [22] consistem num formalismo bastante conveniente à modulação de sistemas concorrentes. Nasceu por meio de Jacques Vautherin [22] a partir do formalismo amplamente conhecido - Petri Nets - através da substituição dos chamados *black tokens* por dados de tipos definidos de forma algébrica (designados por tipos de dados algébricos e abstractos(ADT)). O formalismo contempla uma parte de controlo referente à petri net inerente e uma outra referente aos dados suportados pelos ADT, que por seu turno podem também ser divididos em duas partes: a assinatura que define as operações e as constantes da álgebra envolvida e a axiomatização que fornece a semântica das operações definidas na assinatura.

3.1.1 Visão geral

As APN são utilizadas essencialmente para fins de simulação de sistemas concorrentes e são a base de outros formalismos mais complexos, como o CO-OPN sobre o qual a linguagem SATEL foi desenvolvida. Por este motivo utilizar as APN nesta dissertação é um primeiro passo para se poder estender a abordagem a outros formalismos

mais sofisticados, tendo-se em conta que para este trabalho o objectivo maior é fazer prova de conceito sobre as potencialidades da SATEL, fornecendo-lhe uma possível semântica operacional.

3.1.1.1 Tipos de dados algébricos (ADT)

As Redes de Petri Algébricas, como já referido, recorrem-se de tipos de dados algébricos para representar os seus *tokens* e para poder expressar condições sobre as transições.

Os ADT são formalismos que permitem descrever tipos de dados de uma forma totalmente abstracta. São definidos através de uma assinatura que contém a definição das operações e geradores e por um conjunto de axiomas, que podem incluir formulas algébricas e teoremas. São um formalismo muito útil para representar estruturas de dados porque com ele se define não só a sintaxe do tipo de dados, como também se consegue definir as propriedades de uma forma puramente declarativa e independente de qualquer implementação. É exactamente por este motivo (a independência em relação à implementação) que com o trabalho desta dissertação - geração de testes abstractos - se poderá facilmente realizar futuramente a fase de concretização dos testes para qualquer linguagem.

Listagem 3.1: ADT naturals

```
1  ADT naturals
2  Sort nat
3  Generators
4      zero -> nat ;
5      suc : nat -> nat ;
6  Operations
7      plus : nat, nat -> nat ;
8      sum : nat, nat -> nat ;
9      eq : nat, nat -> bool;
10     lt : nat, nat -> bool ;
11 Axioms
12     sum(X, zero ) = zero ;
13     sum(X, suc(Y))= suc(sum(X,Y)) ;
14     eq(zero, suc(X)) = false;
15     eq(suc(Y), zero) = false;
16     eq(zero,zero) = true;
17     eq(suc(X), suc(Y)) = eq(X,Y);
18     lt (zero, suc(X)) = true;
19     lt (suc(X), zero) = false;
20     lt (suc(X), suc(Y)) = lt(X,Y);
21 Where
22      X : nat ;
23      Y : nat     ;
```

A listagem 3.1 mostra um exemplo de definição de um ADT para os números naturais. Nesta definição começou-se por definir o título do ADT (linha 2), e o Sort (i.e

o tipo) que este ADT define (linha 3). Nas linhas (4-6) definiu-se os geradores do ADT. $zero$ é um gerador de base, e suc é um gerador recursivo, que tem um argumento do tipo *nat*. Com estes geradores, na prática, pode-se construir todos os naturais a partir do valor $zero$, da seguinte forma:

1. $zero$
2. $suc(zero)$
3. $suc(suc(zero)$
4. $suc(suc(suc(zero)$
5. $suc^n(zero)$ ... [1]

De seguida, nas linhas 7 a 11 tem-se a assinatura das operações que pertencem ao ADT. Cada uma das operações é definida através do nome, dos tipos dos argumentos e do tipo do resultado. Da linha 12 à linha 21 têm-se os axiomas sobre as operações do ADT, que definem as propriedades do próprio Sort e que fornecem semânticas às assinaturas anteriormente definidas. Por fim, no final da definição declaram-se os sorts das variáveis utilizadas nos axiomas.

Pode-se observar que é recorrente a utilização do sort bool. Neste trabalho, os ADT podem referenciar sorts já definidos. Neste caso pode-se ver uma definição de várias álgebras no início do apêndice A.3 como parte integrante de um exercício maior explicado nos próximos capítulos.

3.1.1.2 Petri nets algébricas (APN)

Tendo já visto o que são ADTs, como se definem e para que servem, pode-se então relacioná-los com as APN. Atente-se na figura 3.1.

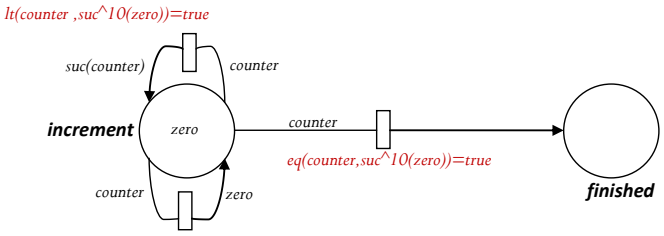

Figura 3.1: Rede de Petri Algébrica de um Contador

[1] Notação utilizada para expressar **n** aplicações sucessivas do gerador **suc**

Na figura tem-se uma APN representada graficamente. Com os círculos representam-se *Places* que contêm um *multiset* (estrutura comparável a um conjunto que pode conter elementos repetidos do mesmo tipo). Este *multiset* contém termos algébricos construídos recorrendo a uma determinada álgebra, cujo sort é também o sort do *place*.

Com as setas definem-se arcos entre os *places* e as transições. Estes arcos contêm pesos, tal como os arcos das Redes Petri tradicionais, sendo que neste caso estes pesos são também *multisets* que contêm termos algébricos à semelhança dos *multisets* dos *places*. Os termos, aqui, podem ser variáveis instanciáveis com qualquer valor ou termos concretizados, gerados a partir dos geradores de um ADT.

Com os rectângulos representam-se Transições que são eventos passíveis de ser disparados quando existem recursos suficientes (no *multiset*) no *place* de origem dos arcos de entrada que possam satisfazer os pesos dos *multisets* destes arcos.

Para que uma transição seja disparada é necessário ainda que as guardas definidas a partir dos axiomas dos ADT, e que estão associadas às transições, sejam também satisfeitas. Após o disparo, o conteúdo do *multiset* do arco de saída, cujos termos livres estão já instanciados, deve ser depositado nos *places* associados aos arcos de saída da transição.

Para o cálculo dos termos de saída e para verificar se as condições são satisfazíveis é muitas vezes utilizada a reescrita de termos [10, 11], com a qual é verificável imediatamente se um determinado termo algébrico corresponde a outro através da redução de ambos por aplicação sucessiva dos axiomas do ADT correspondente, que podem ser aplicados a cada momento.

Após esta explanação pode-se então verificar que a APN da figura 3.1 descreve o comportamento de um contador. A APN contém dois *places* cujo o sort é *nat*(anteriormente definido na listagem 3.1): o *place* **increment** e o *place* **finished**. Inicialmente **increment** contém um *token zero* e **finished** não contém tokens. Neste contador existe uma transição que consome do *place* **increment** um term do tipo nat - counter - e que, caso este term seja inferior a $\{suc^{10}(zero)\}$ $(lt(counter, suc^{10}(zero)) = true)$, então coloca no mesmo *place* o sucessor de counter (*suc(counter)*). Existe ainda outra transição que apenas consome o mesmo *token* - counter - e deposita o token *zero* no mesmo place (equivalente a um reset).

Semanticamente, a escolha de qual destas duas transições é disparada é não determinística, i.e., a dado momento, sendo o token *inferior* a $suc^{10}(zero)$, qualquer uma delas pode ser disparada.

Por fim, existe outra transição, que consome o *token* counter mas que apenas é disparada se esse token for $suc^{10}(zero)$, $(eq(counter, suc^{10}(zero)) = true)$. Nesse caso nenhum *token* é depositado em **finished**, e a APN estagna, não evoluindo para nenhum

estado.

Como se pode observar, uma APN não recebe qualquer tipo de estímulos externos, sendo completamente *'hermética'*, servindo essencialmente para efeitos de simulação, i.e., dada uma marcação inicial obtêm-se os estados sucessivos da APN por aplicação não determinística das transições passíveis de serem disparadas.

Nos trabalhos desta dissertação necessita-se que a APN responda a estímulos externos, e por isso, ir-se-á propôr uma extensão de forma a que isso seja possível.

3.1.2 Semântica

Nesta secção apresentar-se-á uma formalização da semântica das APN, i.e., as condições necessárias para evolução de estado, e a forma como essa evolução se dá. Assim tem-se:

Seja Σ uma assinatura e X um conjunto variáveis. Seja APN uma Petri Net Algébrica. Dada uma marcação M (mapeamento dos nomes dos places para multisets) e uma transição T, T pode ser disparada actualizando a marcação M da APN para uma marcação M', se existe uma substituição σ das variáveis em X, onde:

$$\forall (t = t') \in Cond . \sigma(t) = \sigma(t') \wedge \tag{3.1}$$

$$\left(\sum_{inArc \in T_{in}} \sigma(inArc) \right) \subseteq M \wedge \tag{3.2}$$

$$M' = M[-] \left(\sum_{inArc \in T_{in}} \sigma(inArc) \right) [+] \left(\sum_{inArc \in T_{out}} \sigma(outArc) \right) \tag{3.3}$$

onde:

- $t \in T_{\Sigma,X}$ são termos;
- $\sigma : T_{\Sigma,X} \rightarrow T_{\Sigma,\emptyset}$;
- $\{M, M'\} \subseteq Marcacoes(APN)$;
- T_{in} (resp. T_{out}) é o conjunto de pesos dos arcos de input (resp. output) de T; é ainda um conjunto de termos de $T_{\Sigma,X}$

De modo mais informal tem-se que para disparar uma transição T é preciso encontrar uma substituição das variáveis presentes nas guardas e arcos de input e output da transição, de forma que:

- todas as equações das guardas devem ser satisfazíveis (igualdade entre termo esquerdo e direito)
- a soma de todos os pesos nos arcos de input de T esteja contida na marcação M. A soma é obtida após substituição de todas as variáveis nos arcos de T_{in}, usando a substituição σ. E ainda tendo em conta que [+] é uma soma sobre multisets, e $\subseteq$ é também um operador sobre multisets;
- a nova marcação M' é obtida subtraindo de M o multiset obtido por soma de todos os arcos de input que foram substituídos e subtracção da soma de todos os arcos de output que também foram substituídos; novamente aqui os operadores [-] e [+] são respectivamente operadores de subtracção e soma sobre marcações.

3.2 SATEL

Este capítulo tem o propósito de introduzir a linguagem SATEL (Semi -Automatic TEsting Language) como objecto de estudo primordial ao desenvolvimento da presente dissertação.

3.2.1 Visão geral

A SATEL é uma linguagem de especificação baseada em modelos que foi desenvolvida no âmbito duma tese de doutouramento [17] e tem, originalmente, como objectivo possibilitar a especificação de intenções de teste sobre especificações CO-OPN (Concurrent Object-Oriented Petri Nets) [7] que são utilizadas com o fim de descrever o comportamento dos SUT(através de modelos). Em bibliografia própria, anteriormente referenciada, justifica-se a opção por CO-OPN através da considerável expressividade da mesma visto que inclui não só mecanismos para lidar com concorrência, como também tipos abstractos de dados e encapsulamento através de notações do paradigma orientado a objectos e de módulos de coordenação.

Em particular, estas características conferem ao CO-OPN as qualidades para ser uma boa linguagem de especificação para descrever sistemas distribuídos e concorrentes, que aliada ao paradigma orientado aos objectos possibilita uma abordagem modular.

De um modo mais sucinto, as especificações CO-OPN assentam sobre dois aspectos principais: os tipos de dados definidos algebricamente e o comportamento do SUT (descrito através de Petri nets algébricas que possuem objectos que podem reagir a estímulos externos e, consequentemente, responder com observações através de métodos (*methods*) e portas (*gates*) respectivamente).

Não obstante, como foi já referido na presente dissertação, a especificação das intenções de teste serão feitas sobre especificações APN e não sobre CO-OPN, para a qual a SATEL foi originalmente concebida.

A escolha das APN recai não só sobre o facto de estas consistirem num formalismo mais simplista, com uma semântica mais simples e tratável, mas também porque são o formalismo mais próximo das CO-OPN que permite modelar sistemas concorrentes de forma aceitável.

3.2.2 Descrição da SATEL

Com a linguagem SATEL pode-se definir intenções de teste que se relacionam directamente com as possibilidades de execução (i.e., os diversos comportamentos) ao longo do funcionamento do SUT. Em rigor, cada intenção de teste corresponde a um subconjunto do conjunto de execuções hipoteticamente exequíveis no âmbito do comportamento do sistema (e.g. intenção de teste só sobre o *login* no Sistema) e podem (as intenções) ser especificadas através da indicação da restrição de variáveis que representam diferentes dimensões das fracções do sistema que devem ser testadas.

Do parágrafo anterior pode-se concluir que a SATEL é uma linguagem que serve como *driver* da geração de teste. Pode-se indicar a forma dos caminhos de execução como se de uma expressão regular se se tratasse, acrescentar algumas condições sobre esses caminhos ou sobre variáveis de estímulo e de observação e assim obter testes de uma forma controlada, abrangendo determinada funcionalidade especificamente.

Como se pode observar na gramática e no metamodelo em apêndice, as intenções de teste são expressas através de um conjunto de fórmulas baseadas nas fórmulas HML (Hennessy-Milner [17,24]), às quais foram adicionadas variáveis, e que em [17] passam a designar-se por padrões de execução já que subentendem a descrição da execução de alguma parte do sistema. São fórmulas que permitem ao engenheiro de teste guiar a geração dos testes mediante diferentes critérios, condições e hipóteses, já que a sua sintaxe permite, entre diversos aspectos, definir a configuração dos caminhos de execução, a configuração/padrões das operações do SUT e dos respectivos parâmetros dos métodos e *gates* (*outputs* e *gates*) (a estas possibilidades chamamos hipótese de regularidade). Outras hipóteses oferecidas pela SATEL são a de uniformidade e de subuniformidade aplicáveis sobre variáveis e que se referem à possibilidade de redução dos casos de teste através da selecção, quer de um único valor do domínio das variáveis algébricas (uniformidade) , quer de vários (subuniformidade). Estas duas últimas hipóteses estão contempladas como trabalho futuro desta dissertação.

De uma forma mais informal e com o fim de tornar a compreensão da SATEL mais fácil para o leitor introduzir-se-á a linguagem passo-a-passo começando-se com as

questões mais básicas.

Desta forma, com a SATEL a especificação de um modelo centra-se num ponto fulcral, que são os axiomas das intenções de teste aqui apresentadas com sintaxe similar à proposta, tanto pelo autor como pelos trabalhos desta dissertação.

Tome-se como exemplo, o Counter, introduzido na secção anterior, e faça-se uma extensão de forma a torná-lo minimamente compatível com a notação CO-OPN. Na figura 3.2 pode-se ver que o modelo é o mesmo, mas foram-lhe adicionados os métodos (*tick* e *mark*), e gates (*mark* e *time(Counter)*). Semanticamente, após esta adição, as transições acima só serão disparadas quando se reunirem as condições explicitadas no capítulo anterior sobre a semântica das APN, i.e., se as guardas forem satisfeitas e existir uma subsituição de variáveis, e doravante, se for também recebido um estímulo externo, neste caso, através do método *tick*. Por seu turno, a transição abaixo, uma vez disparada, despoletará o funcionamento da gate *time*, fornecendo dados do interior da APN para fora, neste caso permitindo uma observação do valor da variável algébrica $counter$.

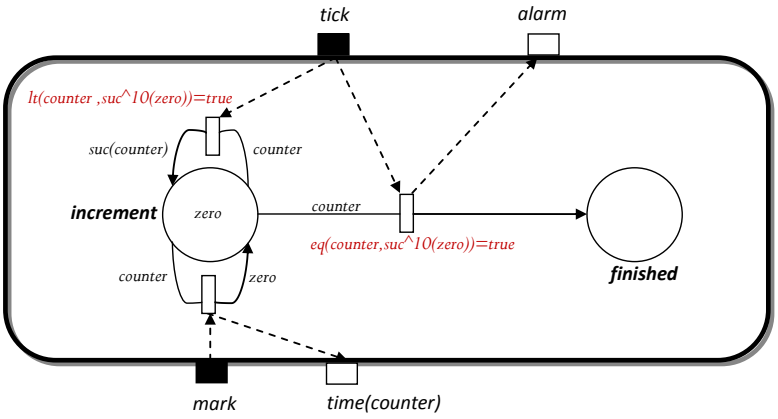

Figura 3.2: Rede de Petri Algébrica Encapsulada de um Contador

Posto isto, tem-se já o necessário para prosseguir com uma introdução informal sobre a linguagem SATEL. Parta-se do princípio que se quer testar o seguinte funcionamento:

- a APN responde a vários estímulos $tick$
- ao fim de n estímulos o valor de $counter$ observado aquando da transição é $suc^n(zero)$ ou $zero$, caso $n = 0$ e nesse momento se por acaso for dado o estímulo $mark$ deve ser feita uma observação $time(counter)$, em que $counter = suc^n(zero)$

Para este caso devem ser gerados testes em que o $counter$ está de acordo com mo-

delo e também testes em que isso não acontece, fornecendo assim um oráculo verdadeiro ou falso, respectivamente.

Para este exemplo, defina-se duas intenções de teste:

TickIntention intenção para gerar caminhos de execução onde se dá o estímulo $tick$ sucessivamente

MarkIntention intenção para gerar o último traço de execução após a geração com a intenção $TickIntention$

Para cada uma das intenções podemos ter diversos axiomas com os quais se exprime a configuração dos caminhos de execução nessa intenção. De uma forma geral um axioma tem a seguinte forma:

$$[\text{Condition}]\text{'}{=>}\text{'Inclusion}$$

A estrutura sugere que se possa lê-la (informalmente) como se de uma implicação se tratasse. À esquerda têm-se condições sobre o axioma que limitam a aplicação do mesmo e que constrangem variáveis. As condições são, como sugere a sintaxe, opcionais. À direita tem-se a Inclusão que é o membro onde se define a configuração do padrão que estamos a gerar, bem como a intenção de teste a que pertence esse padrão.

Observe-se os seguintes axiomas:

- **Top in** TickIntention;
- path **in** TickIntention, nbEvents(path) < 6 =><tick> . path **in** TickIntention;
- $lt(Counter, suc^6(zero)) = true$, path **in** TickIntention => path .<mark **with** time(Counter)> **in** MarkIntention;

O primeiro axioma é um axioma da intenção TickIntention e neste não são especificadas quaisquer condições. Para este axioma apenas se está a gerar o padrão de execução **Top**, ou seja vazio; é o equivalente ao símbolo λ na definição de gramáticas. Portanto, este axioma é, como iremos ver, um auxiliar.

O segundo axioma possui duas condições: exige que a variável de caminho - $path$ - esteja instanciada com padrões de execução gerados por um axioma pertencente à intenção TickIntention e requer que o número de eventos desse padrão seja inferior a 6. Satisfeitas as condições, este axioma gerará padrões em que concatena o padrão $path$ com uma sincronização, cujo estímulo é o evento $< tick >$. Note-se que o operador ponto (.) é o operador de concatenação de caminhos de execução.

Estes dois axiomas pertencem ambos à intenção TickIntention. Numa analogia, em termos de funções recursivas, o primeiro é um caso de base e o segundo é um axioma recursivo, já que requer que a variável $path$ esteja instanciada com padrões da intenção TickIntention e a partir disso gera mais padrões para essa intenção de teste.

Em termos práticos estes dois padrões juntos gerariam os seguintes traços:

- <tick>.Top
- <tick>.<tick>.Top
- <tick>.<tick>.Top
- <tick>.<tick>.Top
- <tick>.<tick>.<tick>.Top
- <tick>.<tick>.<tick>.<tick>.Top
- <tick>.<tick>.<tick>.<tick>.<tick>.Top

(Note-se que apenas seis eventos são admitidos pelo segundo axioma, dada a segunda condição, em relação à variável path.)

O terceiro axioma não é recursivo. Este exige que a variável $path$ contenha padrões na intenção $\mathrm{TickIntention}$ (lado esquerdo do axioma), para a partir disso gerar padrões de execução para a intenção de teste $\mathrm{MarkIntention}$ (lado direito do axioma).

Aproveitando este axioma para introduzir a noção de $\mathrm{ConditionAtom}$, neste axioma, tem-se portanto dois $\mathrm{ConditionAtom}$, sendo o segundo um $\mathrm{Inclusion}$ (uma restrição sobre o domínio da variável de caminho). Às variáveis de caminho atribuímos o tipo $\mathrm{HMLPrimitive}$.

Os $\mathrm{Inclusion}$ são, como se pode ver, ambivalentes: servem como $\mathrm{ConditionAtom}$ e constituem também o lado direito do axioma.

O primeiro $\mathrm{ConditionAtom}$ é uma igualdade algébrica ($\mathrm{AlgEqualtity}$) e é uma restrição sobre uma variável algébrica (neste caso do sort nat). Serve para obrigar a que a igualdade $lt(Counter, suc^6(zero)) = true$ seja verdadeira (i.e., que se possa reescrever o termo do lado esquerdo da equação no do lado direito).

Com a aplicação deste axioma seriam concatenados aos padrões já gerados pelos dois primeiros axiomas o padrão <mark **with** time(Counter)>, em que a variável Counter seria resolvida para os valores do conjunto $\{zero, suc^n(zero)\}$ com $n \in \{0..5\}$. Assim sendo, obter-se-iam as seguintes soluções finais:

- <tick>.Top.<mark with time(Counter)
- <tick>.<tick>.Top.<mark with time(Counter)
- <tick>.<tick>.Top.<mark with time(Counter)
- <tick>.<tick>.Top.<mark with time(Counter)
- <tick>.<tick>.<tick>.Top.<mark with time(Counter)
- <tick>.<tick>.<tick>.<tick>.Top.<mark with time(Counter)

- <tick>.<tick>.<tick>.<tick>.<tick>.Top.<mark with time(Counter)

Por definição, uma solução final é uma solução derivada a partir de um axioma que gera testes para uma intenção de teste que não é referenciada por nenhum ConditionAtom(Inclusion) dos restantes axiomas, ou que o sendo, seja pelo próprio axioma. Nesse caso o axioma é recursivo. Se se considerar um grafo onde cada vértice representa um axioma e os respectivos arcos orientados representam a dependência entre os axiomas, então as soluções finais serão as que forem geradas a partir de um axioma, cujo vértice não aceita arcos de entrada, i.e. é um vértice fonte. Visualmente, para o exemplo em análise, a figura 3.3 mostra que o terceiro axioma produz os testes finais.

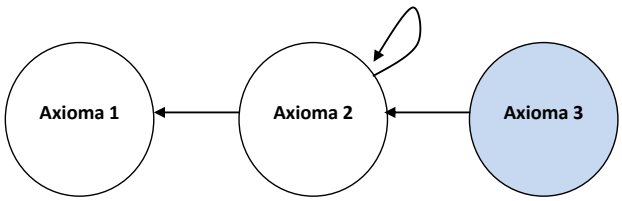

Figura 3.3: Grafo de dependências entre axiomas

Finalmente, após a geração dos testes o último passo seria obter os oráculos para cada um dos testes. Assim ter-se-iam alguns testes que deverão passar e outros não. Mostra-se em seguida um subconjunto reduzido de alguns testes e respectivo oráculo.

Verdadeiro: <tick>.<mark with time(suc(zero))>

Verdadeiro: <tick>.<tick>.<mark with time(suc(suc(zero)))>

Falso : <tick>.<tick>.<mark with time(suc(suc(suc(suc(zero)))))>

Note-se que nos testes cujo oráculo é verdadeiro, o número de eventos tick corresponde ao valor da variável Counter devolvida pela *gate* time.

3.2.3 As construções suportadas pela linguagem SATEL e semântica

Depois de introduzida a linguagem sob um ponto de vista geral, neste capítulo enumeram-se as construções que a linguagem permite. Nos apêndices A.1 e A.2 encontra-se, respectivamente, o metamodelo correspondente à gramática do SATEL e a sintaxe concreta escolhida para os trabalhos desta dissertação e que tem por base a gramática proposta originalmente. Nos apêndices, tanto o metamodelo como a gramática apresentados foram desenvolvidos na plataforma de modelação do Eclipse [2], e contêm as

[2]http://www.eclipse.org

construções adicionais (eg. ADT, APN), que serão discutidas nos próximos capítulos. Aqui apresenta-se apenas as que a SATEL suporta.

Assim sendo, a linguagem por defeito, como ConditionAtom, aceita as seguintes construções:

ConditionAtom	Descrição
AlgEquality	Igualdade entre dois termos algébricos
SyncEquality	Igualdade entre dois termos de sincronização
HMLEquality	Igualdade entre dois termos HML(padrões de execução)
BooleanEquality	Igualdade entre dois termos Booleanos
ArithmeticEquality	Igualdade entre dois termos aritméticos
Not	Negação de um termo booleano
Positive	Padrão de execução positivo
Sequence	Padrão de execução é uma Sequencia
Trace	Padrão de execução é um traço
BooleanVariable	Variável booleana
BooleanValue	Valor booleano
BooleanAnd	Conjunção de dois termos booleanos
BooleanOr	Disjunção inclusiva de dois termos booleanos
BooleanEquals	Igualdade entre dois termos aritméticos
BooleanNotEquals	Desigualdade entre dois termos aritméticos
BooleanGT	Desigualdade "Maior"entre dois termos aritméticos
BooleanLT	Desigualdade "Menor"entre dois termos aritméticos
BooleanGTE	Desigualdade "Maior ou Igual"entre dois termos aritméticos
BooleanLTE	Desigualdade "Menor ou Igual"entre dois termos aritméticos

Tabela 3.1: ConditionAtom's aceites pela linguagem SATEL

Desta tabela (3.1) há alguns ConditionAtom que necessitam de explanação adicional, nomeadamente *Positive*, *Trace*, e *Sequence* que têm como argumento, um *HMLTerm*. Um *HMLTerm* é padrão de execução constituído por uma lista de fórmulas e/ou variáveis ordenadas que veremos na tabela 3.3. Em rigor já vimos pelo menos duas: uma explícita *HMLTop* e outra implícita pelo operador de concatenção de caminhos (ponto); trata-se da fórmula *HMLNext*. E existem ainda duas outras fórmulas: *HMLNot* e *HMLAnd*.

Informalmente, um *HMLNext* é um construtor de caminho que possui dois argumentos. Um evento e uma fórmula seguinte. É portanto um operador que permite sequenciar caminho. Por seu turno, um *HMLNot* é um construtor de caminho pela negativa que tem um *HMLTerm* como argumento, i.e., quando aparece um *HMLNot* no padrão de execução significa que se pretendem todos os caminhos que não contêm o argumento desse *HMLNot* a partir do ponto em que este aparece. A figura 3.4 retrata essa situação.

Em relação ao construtor *HMLAnd*, este, por seu turno, permite construir caminhos que possuem ramos de execução diferentes. Num caminho que possua um *HMLAnd*, o traço de execução nesse ponto é decidido não deterministicamente por um dos ramos do *HMLAnd*.

Posto isto, pode-se já entender o que são os *ConditionAtom* atrás referidos: *Positive*,

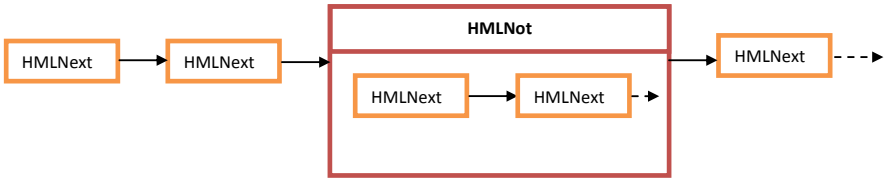

Figura 3.4: Visualização de um HMLTerm contendo fórmulas HMLNot

Trace, e *Sequence*. *Positive* tem valor de verdade quando o seu argumento não possui fórmulas negativas *HMLNot*. *Sequence* tem valor de verdade quando o seu argumento não possui *HMLAnd*. E por fim, *Trace* tem valor de verdade quando para o seu argumento, os *ConditionAtom*, *Positive* e *Sequence* têm também valor de verdade. De modo mais informal, um caminho é um traço de execução se não contém negações e bifurcações. Mais se acrescenta que para esta dissertação o interesse recai sobre caminhos que sejam traços, sendo que os que não são, estão direccionados para trabalho futuro, a partir desta tese.

Em relação aos *ArithmeticTerm*, que são argumentos de alguns *ConditionAtom* da tabela 3.1, a linguagem SATEL define os seguintes presentes na tabela 3.2.

ArithmeticTerm	Descrição
IntegerVariable	Variável Número Inteiro
IntegerValue	Valor Inteiro
BOPPlus	Soma de dois Inteiros
BOPMinus	Subtracção de dois Inteiros
BOPTimes	Multiplicação de dois Inteiros
BOPDiv	Divisão Inteira de dois Inteiros
NBEvents	Medida de dimensão de caminho: Número de eventos.
Depth	Medida de dimensão de caminho: Profundidade.
UOPMinus	Negação unária sobre inteiros

Tabela 3.2: ArithmeticTerm's aceites pela linguagem SATEL

Pensa-se que os *ArithmeticTerm* aceites pela linguagem sejam auto-explanatórios, uma vez que a maioria são operações básicas sobre inteiros. Destinguem-se de entre todos o *NBEvents* e o *Depth*. Semanticamente, o *NBEvents* deve retornar o número de eventos de um padrão de execução segundo as seguintes regras informais:

Percorrendo um caminho de execução, se se encontrar uma fórmula:

HMLTop: não é contabilizado nenhum evento.

HMLNext(Event(Synchronization(Input,Output)), h:HMLFormula): é contabilizado um evento por cada sincronização de input e somam-se ainda os eventos de **h**

HMLNot: são contabilizados os eventos do seu argumento

HMLAnd: é contabilizada a soma dos eventos de ambos argumentos

Por outro lado o *Depth*, semanticamente, deve retornar a profundidade de um padrão de execução, i.e, o comprimento do maior ramo do padrão de execução. Deve ser calculado segundo as seguintes regras informais:

Ao se percorrer um caminho de execução, se se encontrar uma fórmula:

HMLTop: nada é contabilizado.

HMLNext(Event), h:HMLFormula): o Depth é incrementado de uma unidade e soma-se ainda o Depth de **h**

HMLNot: é contabilizada a profundidade do seu argumento

HMLAnd: é contabilizado o máximo entre os *Depth* de cada argumento

Por fim a tabela 3.3 mostra as construções que permitem definir um padrão de execução.

HMLTerm ou relacionado	Descrição
HMLTerm	Padrão de execução
HMLNext	Fórmula de padrão de execução que permite definir um evento e uma fórmula seguinte
HMLTop	Fórmula de padrão de execução equivalente a vazio
HMLNot	Fórmula de negação de padrão de execução
SynchronizationInputTerm	Referente a um método de input e respectivos parâmetros algébricos
SynchronizationOutputTerm	Referente a uma *gate* de output e respectivos variáveis algébricas de output.
HMLEvent	Um evento constituído por um SynchronizationInputTerm e *opcionalmente* SynchronizationOutputTerm

Tabela 3.3: ArithmeticTerm's aceites pela linguagem SATEL

Com a linguagem SATEL pode-se, como se viu, definir variáveis do tipo Inteiro ($\mathrm{PrimitiveInteger}$ e $\mathrm{Boolean}$) e de caminho de execução ($\mathrm{PrimitiveHML}$). Adicionalmente podem ainda ser definidas variáveis de estímulo - $\mathrm{PrimitiveStimulation}$ - e de observação ($\mathrm{PrimitiveObservation}$), utilizáveis nos padrões $<\mathrm{input\ with\ output}>$ e, por fim, variáveis algébricas de um determinado *sort*.

Aquando da resolução das variáveis, a linguagem ainda oferece duas construções unárias interessantes: hipótese de **uniformidade** e hipótese de **subuniformidade** sobre qualquer variável. Quando aplicado um predicado de uniformidade sobre uma variável deve ser devolvido um valor aleatório disponível para essa variável, dadas as restrições sobre ela incidentes. Por outro lado, quando aplicada a hipótese de subuniformidade deve ser escolhido um valor por cada comportamento despoletável pelo método de input envolvido na intenção de teste, dado o valor dessa variável. A título exemplificativo, se hipoteticamente a variável *counter* estivesse explicitamente implicada no segundo axioma do exemplo, que se tem vindo a explorar, tal como está implicada no terceiro axioma, aplicar uma hipótese de subuniformidade sobre essa variável resultaria na escolha de apenas dois valores para a instanciar: o valor $suc^{10}(zero)$

e qualquer outro valor diferente que respeitasse a restrição $lt(counter, suc^{10}(zero)) = true$.

Apenas seriam seleccionados estes dois valores, pois são os suficientes para cobrir os dois possíveis comportamentos da APN; ou transita para o place $\mathrm{finished}$ ou incrementa o valor de counter no place $\mathrm{increment}$.

3.3 DSLTrans - Transformação de modelos

Embora não constitua um foco desta dissertação, a transformação de modelos acabou, após diferentes tentativas de implementação a explicitar nos próximos capítulos, por se revelar uma técnica útil e fundamental para a solução proposta. Existem diversas ferramentas que utilizam diferentes técnicas para transformar modelos conformantes com determinado metamodelo A em modelos conformantes com determinado metamodelo B. Metamodelos estes que podem ser o mesmo (no caso das transformações endógenas) ou diferentes (transformações exógenas).

São exemplos do enorme grupo de ferramentas, o ATOM3[3], o ATL[4] (da plataforma Eclipse), FUJABA [5] e o DSLTrans[6] que é a principal ferramenta adoptada no desenvolvimento da presente dissertação.

A ferramenta DSLTrans[7] [4] [18] é uma ferramenta de transformação de modelos desenvolvida pelo grupo de investigação onde se insere esta dissertação. A ferramenta permite fazer transformações sendo apenas necessário indicar com regras qual o padrão que se pretende encontrar no modelo e qual o padrão que se pretende construir a partir desse. As regras estão organizadas em diferentes *layers* o que, semanticamente, impõe que regras de *layers* sequenciais sejam aplicadas pela ordem em que aparecem as respectivas *layers*.

A técnica usada permite que apenas se especifiquem as regras necessárias para reconhecer os padrões que nos interessam, ao contrário do que acontece por exemplo com o ATL, cujas regras, são processadas de forma indutiva, sendo nesse caso necessário descrever regras para cada uma das classes que sejam containers directos ou indirectos das classes que nos interessam para os padrões. No ATL, o único caso em que não é necessário processar toda a sub-árvore até ao nó do padrão que interessa é nas transformações endógenas em modo de refinamento, que serve essencialmente para fazer manipulações no próprio modelo de entrada. Para uma visão mais detalhada

[3] http://atom3.cs.mcgill.ca
[4] http://eclipse.org/atl
[5] http://www.fujaba.de
[6] http://pessoa.fct.unl.pt/rlf18343/dsltrans_manual.pdf
[7] http://solar.di.fct.unl.pt/twiki/pub/BATICCCS/ReleaseFiles/dsltrans.october.2010.zip

atentemos no metamodelo da figura 3.5.

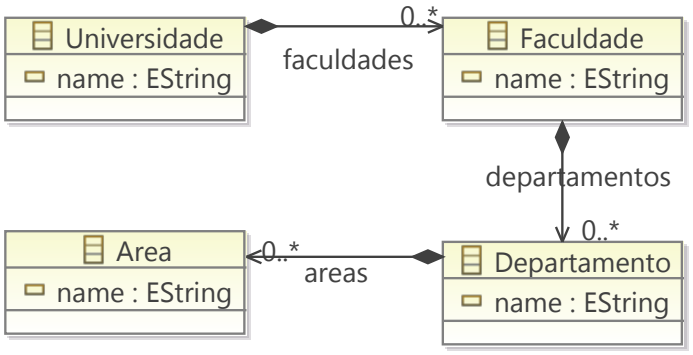

Figura 3.5: Metamodelo Universidades

Numa transformação em que apenas se quer encontrar todos os departamentos de um modelo, se esta for implementada no ATL é necessário definir regras para que se mapeem os objectos Universidade e Faculdade em entidades do metamodelo destino, por outro lado no DSLTrans apenas definimos uma regra cuja execução fará a identificação/*matching* de todos os departamentos e que o mapeará em alguma entidade no metamodelo de destino.

Relativamente às regras do DSLTrans, e reforçando a bibliografia do DSLTrans anteriormente apresentada, veja-se a figura 3.6 e considere-se a existência de um metamodelo destino minimalista que apenas contém uma classe 'Unidade' contida por algum objecto e que apenas tem 'name' como atributo.

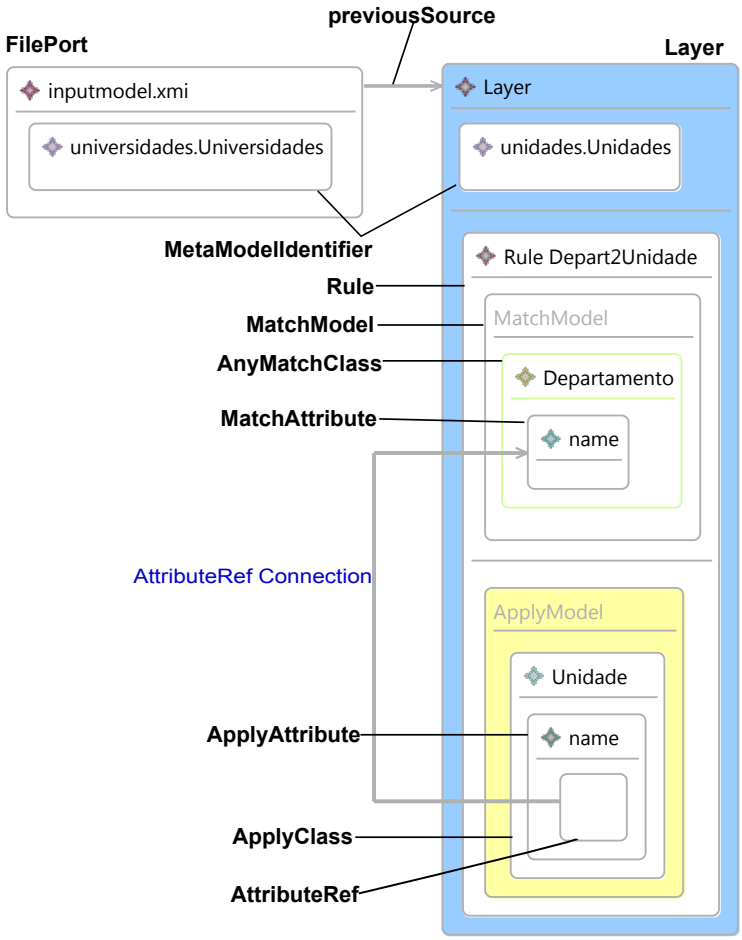

Figura 3.6: Regra de transformação em DSLTrans

Através da figura notemos que é necessário indicar o modelo e metamodelo de entrada em FilePort, e que a ele está ligada a primeira *layer*(a azul) pela relação previousSource. Em cada *layer* é também necessário indicar o metamodelo da instância que resultará da transformação. Isto é feito através de elementos MetaModelIdentifier.

Em cada *layer* podemos ter várias regras e cada regra contém duas secções: uma de *match* (MatchModel) e uma de *apply* (ApplyModel).

Na secção de *match* colocam-se as classes dos objectos que queremos transformar bem como seus atributos e relações. Na secção de *apply* podemos fazer exactamente o mesmo através dos objectos adequados à secção ApplyModel tais como ApplyClass e ApplyAttribute. Adicionalmente podemos reutilizar valores que foram identificados na parte de *match* para construir objectos na parte de *apply*.

Podemos ainda referenciar objectos já criados em alguma regra de *layers* anteriores desde que devidamente identificados com um ApplyAttribute, que deve obrigatoriamente designar-se por 'ApplyAttribute' ou simplesmente não ter designação. Este tipo de objecto deve ter uma restrição (BackwardRestriction) para a classe que o possa

ter gerado. Por exemplo, relativamente à figura, o objecto Unidade que estamos a criar poderia hipoteticamente ter sido criado numa *layer* anterior a partir de um objecto Departamento e, nesse caso, poderíamos indicar que queríamos usar o mesmo objecto desde que ambos, nas duas regras, tivessem um ApplyAttribute com o Atom contendo um identificador com o mesmo valor em ambas. Adicionalmente pôr-se-ia um BackwardRestriction da Unidade para o objecto Universidade.

É ainda possível no DSLTrans identificar padrões de associação entre classes que se relacionam tanto directamente como indirectamente.

Para finalizar consegue-se através do DSLTrans especificar padrões pela negativa e existenciais, isto é, temos objectos(AnyMatchClass, ExistsMatchClass e NegativeMatchClass) e relações (NegativeMatchAssociation e AplyAssociation) que, conjuntamente, permitem expressar padrões como *"Todos os departamentos sem faculdades"*, ou *"Todos os departamentos que contenham pelo menos uma faculdade"*.

Para mais informação sobre o DSLTrans deve-se consultar a referência. Não obstante, no contexto da descrição da solução proposta conseguir-se-á explicitar de melhor forma o desenvolvimento de modelos de transformação via DSLTrans.

3.3.1 DSLTrans - Sintaxe concreta textual

O DSLTrans foi inicialmente desenvolvido com uma notação visual e, embora esta seja a melhor forma de se observar um modelo transformacional e extrair a sua semântica de forma praticamente imediata, foi desenvolvida no âmbito desta dissertação, uma linguagem textual com o fim de disponibilizar uma sintaxe diferente (ao estilo das linguagens de programação generalistas) e também para reduzir o enorme volume gráfico da transformação apresentada.

A gramática desta sintaxe foi desenvolvida recorrendo à ferramenta EMFText (da plataforma Eclipse) e encontra-se no apêndice A.7. De uma forma geral resulta em transformações em que se omite muito do detalhe inerente ao metamodelo do DSLTrans.

Vejamos a listagem 3.2 extraída da solução que apresentaremos no próximo capítulo. É inicialmente definido o fileport que será o previous da *layer* "Entities". Para ambos é definido o metamodelo indicando o nome e o respectivo filepath. Dentro da *layer* estão as regras que, à semelhança do que acontece na sintaxe visual, contêm uma parte de *match* e uma de *apply*. Na parte de *match* a regra apresentada contém duas classes identificadas pelo metamodelo antes de '::' e pelo nome da classe depois de '::'.

Listagem 3.2: Exemplo em DSLTrans textual

```
1
2 File
```

```
3       id = input
4       uri = 'timermodel.SATEL'
5
6       metamodel(
7                   mmname = SATEL.SATEL
8                   uri = 'SATELAPN.ecore'
9               )
10
11  def 'Direct Mappings' : layer  'Entities'
12      previous = 'input'
13      output = "result.mprologTR"
14      metamodel(
15              mmname = mprologTermReference.MprologTermReference
16              uri = 'mprologTermReference.ecore'
17          )
18
19      rule 'IntegerVariable'
20          match with
21              m98:
22                      any SATEL.AlgebraicExpressions.arithmeticterms::IntegerVariable
23              m99:
24                      any SATEL.VariableDeclarations::PrimitiveIntegerVarDec( a100 : name )
25
26              subject to
27                  m98 --(integerVariable)-> m99
28
29
30          apply
31              m101:
32                      mprologTermReference::Variable(
33                          name= sameAs(a100)
34                          self = 'ArithmeticFunctor'
35                      )
36      end rule
```

Estão ainda etiquetadas por um identificador (m98 e m99) que pode ser usado para as referenciar noutras partes da regra. A classe PrimitiveIntegerVarDec contém ainda um atributo no padrão de *match* que está também etiquetado.

Ambas as partes contêm uma secção de restrições ('subject to') que permite especificar relações entre as classes. No caso do *match model* forçamos a que as classes referenciadas m98 e m99 estejam relacionadas directamente através da relação integerVariable. Ainda na secção de restrições podemos utilizar as construções:

A!-(<relationname>)->B para expressar que A não deve estar relacionado com B através de <relationname> ;

A∼∼(?)∼>B para expressar que A deve estar relacionado com B não necessariamente directamente. Pode conter objectos de outras classes pelo meio do ramo que liga A a B. Mais formalmente, dir-se-á que B deve ser alcançável a partir de A. O nome da relação não acrescenta semântica ao modelo, pode ser indicado para fins de legibilidade.

A!~(<relationname>)~>B para expressar que B não deve ser alcançável a partir de A de forma directa ou indirecta.

Deve notar-se ainda que, no match model, ao invés de se utilizar as construções *any*, pode-se utilizar também as construções *exists* e *not* para representar padrões negativos.

Voltando ao exemplo, na secção de *apply* criamos o objecto etiquetado m101: mprologTermReference::Variable com o atributo name cujo valor deve ser o mesmo do que foi encontrado no padrão de *match* e que está etiquetado com a100. É ainda criado um atributo self com o valor definido pelo utilizador. Este é o ApplyAttribute referido na secção anterior que permite que esta mprologTermReference::Variable seja referenciada futuramente noutra regra de uma *layer* subsequente.

Repare-se que a etiquetação foi um mecanismo criado que não pertence ao metamodelo do DSLTrans e que tem como finalidade tornar possível a resolução de referências nos editores gerados pelo EMFText. Simultaneamente facilita a especificação dos modelos de transformação, uma vez que se assemelha à declaração de variáveis das linguagens textuais em geral.

Para finalizar vejamos a listagem 3.3, também extraída da solução a apresentar.

Listagem 3.3: Excerto da regra para transformar uma APN

```
1   ...
2     rule 'For  APN creates a Clause with Head'
3     match with
4         m559: any SATEL.APN.apnmm::APN
5         m560:    any SATEL::Model
6
7         subject to
8             m560 ~~(contains)~> m559
9
10    apply
11        m561: mprologTermReference::Functor(
12                    self = 'InitialMarcationFunctor'
13                    )
14        m564: mprologTermReference::Head
15                    subject to
16
17        subject to
18            m564 --(ownedFunctor)-> m561
19
20    restrictions
21        m561 derived from m559
22
23    end rule
24  ...
```

Com esta listagem pretende-se mostrar a utilização de associações indirectas (linha 8), a parte de "subject to" da secção *apply* (linha 17), e por fim na linha 20 as restrições adicionais. Estas restrições funcionam da seguinte forma neste contexto particular:

Indicando que m561 deve ser derivado de m559 supõe-se que há uma regra numa *layer* anterior cujo padrão de *match* contém uma classe SATEL.APN.apnmm::APN e cujo padrão de *apply* contém uma classe mprologTermReference::Functor que contém um ApplyAttribute identificado com ***self*** = *'InitialMarcationFunctor'* que é o mesmo usado no objecto m561. Desta forma indicamos que o objecto m561 deve ser exactamente o mesmo que já foi gerado noutro contexto em que apareceu a mesma classe SATEL.APN.apnmm::APN.

A este tipo de restrições (as do exemplos) designamos de PositiveBackwardRestriction. O DSLTrans possui também as NegativeBackwardRestriction que têm semântica contrária à supra-explanada, isto é, o objecto não deve ser o mesmo que foi gerado anteriormente. A sintaxe textual no caso negativo seria m561 not derived from m559.

3.3.2 Conclusões

Nesta secção apresentou-se a linguagem SATEL, as diversas construções e uma semântica informal das mesmas. Em rigor, a semântica da linguagem SATEL foi especificada recorrendo a álgebras para cada entidade aqui apresentada: álgebras para os padrões de execução, para os *ConditionAtom*'s(*BooleanTerms*), para os *ArithmeticTer*ms, e também para os estímulos e observações. Nestas álgebras, que definem toda a linguagem SATEL, foram definidos axiomas que permitem formalmente especificar tudo o que foi explanado neste capítulo. Para uma melhor compreensão da semântica, em termos formais, é sempre possível consultar a referência bibliográfica [17](p149-p157).

Em relação ao DSLTrans, foi introduzida a linguagem, a forma de utilização e uma linguagem textual criada no contexto desta dissertação. Tentou-se introduzir as idiosincrasias do DSLTrans de forma faseada com o fim de as clarificar à medida que foi introduzida a descrição e as diferentes sintaxes. Conclui-se justificando que foi decidido utilizar o DSLTrans não só porque foi desenvolvida no âmago do grupo em que o trabalho se insere, mas mais fortementemente por ser uma linguagem em que apenas é necessário definir as regras para as entidades que interessam (em oposição ao ATL, por exemplo) e também porque os modelos de transformação resultantes são declarativos e executáveis, validando a transformação *per se*; o que novamente em oposição ao ATL (que se recorre de construções iterativas/imperativas) causa uma perda de legibilidade e correspondência com a especificação do modelo.

4

Abordagem proposta/Solução

Neste capítulo pormenorizar-se-á a solução apresentada no capítulo 1. Serão apresentados exemplos e justificadas as opções tomadas ao longo das diversas etapas. O exemplo introduzido no capítulo anterior - o *timer* - será permanentemente referenciado para que o leitor consiga seguir melhor as diversas etapas da solução.

É importante, desde já, notar que o metamodelo utilizado - SATEL⊕EAPN - apresentado no apêndice A.1, reutiliza uma parte do metamodelo do AlPiNa. Essa parte é a correspondente ao *sub-package* APN do metamodelo(linha 338). Esta reutilização e extensão foram feitas visando a possível integração futura da ferramenta proposta nesta tese com o *model-checker* AlPiNa. Por esse motivo este metamodelo não foi alterado, mas sim, apenas estendido ou truncado para que a compatibilidade entre modelos esteja garantida.

4.1 Decisão sobre a abordagem implementada

Em primeira análise, uma implementação do SATEL poderia, à semelhança da implementação de muitas outras linguagens, ter simplesmente sido implementada por uma linguagem imperativa como o JAVA. De facto, implementou-se esta solução (apêndice A.9.9), no entanto rapidamente se constatou que esta apresentava problemas sérios que comprometiam o pretendido.

Para esta primeira abordagem, em que não havia ainda linguagem de suporte ao SATEL⊕EAPN, era feito *parsing* dos modelos da plataforma Eclipse, era percorrida a árvore do modelo e eram recolhidos dados com variáveis recursivas, não recursivas, e

de caminho, entre outra informação, de acordo com um algoritmo desenvolvido para o efeito. Eram ainda criadas tabelas para cada uma das intenções de teste, com as quais se pretendia, no final do processo, obter todas as soluções que fossem válidas.

Nesta solução os axiomas da intenção de teste eram exercitados consoante um grafo de dependências estabelecido entre estes. Embora parecendo uma solução promissora, ela pecava em três aspectos: o esquema de tabelas tornava, por vezes, a resolução de variáveis de caminho, com vários níveis de entrosamento impraticável. A resolução de condições com diversas variáveis também não podia ser feita visto que no momento em que o seu valor devia ser necessário, estas ainda se encontravam por instanciar (idiossincrasia do algoritmo e do processamento axioma a axioma). Por fim, consistia num tipo de implementação que, por usar uma linguagem de âmbito geral, não só tornava difícil a validação do SATEL enquanto ferramenta MBT, como não era possível relacionar directamente a semântica denotacional com a implementação, contribuindo também para uma validação ineficiente.

Uma segunda abordagem, igualmente implementada consistia em tirar partido da semântica denotacional do SATEL. Tal como foi mencionado no capítulo anterior, em [17] foram definidas diversas álgebras para cada uma das entidades usadas no SATEL (padrões de execução, $\mathrm{ConditionAtom}$, $\mathrm{ArithmeticTerm}$,...), que permitiram definir a sua semântica e a semântica dos operadores. Nesta segunda implementação utilizava-se o DSLTrans (introduzido no capítulo 3.), mas apenas para transformar ADTs, e reaproveitava-se a transformação automática para transformar a própria especificação dos modelos SATEL$\oplus$EAPN em cláusulas Prolog, observando a parte SATEL do modelo como um ADT, com axiomas, operações e geradores.

O problema nesta abordagem tornou-se evidente na sua implementação, pois verificou-se que existia diferença entre os níveis de abstracção entre os ADTs usados na especificação das àlgebras que constituem a semântica denotacional do SATEL e os ADTs aceites pela linguagem SATEL como *input*. Esta diferença de abstracção, em que o primeiro se encontra num nível superior, torna impossível a utilização da mesma transformação em ambos os casos, e ainda que se tentasse concretizar um pouco as álgebras da semântica do SATEL até ao nível das álgebras de *input*, acabar-se-ia por continuar a abrir mão da proximidade entre implementação e semântica denotacional. Perder-se-ia eficiência por que seria necessário encapsular todos os predicados derivados da transformação e, em geral, abonar-se-ia em favor da complexidade global.

Finalmente, numa terceira abordagem para a implementação do SATEL, que demonstrou ser a mais adequada e que se vai detalhar em seguida, consistiu naquela em que, apesar de se perder generalidade da transformação (deixando de haver uma transformação apenas para ADTs), se transforma cada parte dos modelos SATEL$\oplus$EAPN de

uma forma dedicada, incrementando a eficiência, potenciando implementação futura de heurísticas em determinados casos (que não seria possível se a transformação fosse genérica) e obtendo uma implementação próxima da semântica denotacional.

Entrosadamente com a terceira e última abordagem, iniciou-se a implementação em Java da mesma transformação apresentada neste capítulo. Esta abordagem foi também declinada não só por uma das razões que levou a abandonar a primeira abordagem (distância em relação à semântica denotacional), como também pelo facto de que não havendo modelos explícitos da transformação, a evolução, manutenção, certificação, validação, portabilidade e a legibilidade ficariam comprometidas.

4.2 Extensão das APN

Viu-se no capítulo anterior, nas secções sobre SATEL e APN, que a SATEL foi desenvolvida sobre modelos CO-OPN e que por isso é uma linguagem para especificar intenções de teste para sistemas não só concorrentes como também reactivos, que respondem a estímulos externos e que permitem observar através de *gates*, informação sobre o estado em que se encontram. Viu-se também que as APN foram o formalismo escolhido para ser composto com a SATEL (ao invés do CO-OPN) e que na realidade não suporta reacção a estímulos externos, servindo essencialmente para simulação. Assim sendo concluiu-se que era necessário dotar/estender as APN com a possibilidade de responder a esses estímulos e de permitir a observação da reacção aos mesmos.

No mesmo capítulo, com o fim de introduzir a SATEL, sem entrar na complexidade do CO-OPN, sugerimos um exercício em que na realidade dotámos empiricamente uma APN com estímulos (*methods*) e observações (*gates*), e é exactamente nessa base que se formaliza a extensão das APN. Assim, uma EAPN é uma APN dotada de *gates* de *output* e *methods* de *input*, ambos parametrizáveis algebricamente. Desta forma, uma APN passa a ser uma entidade reactiva e comporta-se como um módulo *stand-alone* do formalismo CO-OPN.

Semanticamente e de modo informal, uma EAPN possui todas as características apresentadas na secção 3.1.2 do capítulo anterior, acrescidas do facto de o disparo de uma transição ser agora efectuado apenas quando também é invocado um método de *input* que lhe esteja associado. Adicionalmente despoleta também as observações nas *gates* que lhe estejam associadas.

Podemos observar esta extensão no *sub-package* APN do metamodelo da SATEL. Observe-se as linhas 9 e 10 da listagem 4.1, onde são adicionados *gates* e *methods* à APN, e as linhas 4 e 5 da listagem 4.2 onde se pode ver que a cada transição estão associados um $\mathrm{MethodCall}$ e várias $\mathrm{GateCall}$ (Invocações de Métodos e de *Gates*).

Listagem 4.1: Metamodelo SATEL - Classe APN

```
1  class APN extends environmentmm::Environment
2  {
3      property ownedPlaces : Place[+] {
           composes };
4      property ownedArcs : Arc[*] { composes
           };
5      property ownedVariables : adtmm::
           Variable[*] { composes };
6      attribute name : String[1];
7      property methods : Method[*] { composes
           };
8      property gates : Gate[*] { composes };
9      property ownedTransitions : Transition
           [+] { composes };
10 }
```

Listagem 4.2: Metamodelo SATEL - Classe Transition

```
1  class Transition extends Node
2  {
3      property ownedGuard : guardmm::Guard[?]
           { composes };
4      property gateCalls : GateCall[*] {
           composes };
5      property methodCall : MethodCall[?] {
           composes };
6  }
```

4.2.1 Composição das EAPN com a SATEL

Na secção anterior viu-se como foram estendidas as APN e localizámos no metamodelo SATEL essa mesma extensão. Implicitamente acabou-se por ver também como foi feita a composição da SATEL com as EAPN. Na verdade se atentarmos ao metamodelo do SATEL, na linha 1 da listagem 4.3 vemos que um HMLNext é composto por um HMLEvent, e que esse HMLEvent possui um inputTerm:**SynchronizationInputTerm** e um inputTerm:**SynchronizationOutputTerm**, que por sua vez podem ser SynchronizationEventInputTerm (resp. SynchronizationEventOutputTerm) ou uma variável WPrimitiveStimulationVarDec (resp. WPrimitiveObservationVarDec) instanciável com qualquer MethodCall ou GateCall.

A composição da SATEL com EAPN foi feita exactamente neste ponto. Tanto SynchronizationEventInputTerm como SynchronizationEventOutputTerm têm uma referência *event* que pode ser respectivamente um InputEvent ou um OutputEvent. Assim, com o fim de compor ambas as linguagens tem-se que:

- um InputEvent da SATEL é um Method das EAPN
- um OutputEvent da SATEL é uma Gate das EAPN

Mais ainda, formalmente tem-se que:

- $(\forall_G\ G \in \mathcal{P}(SATEL.APN.Gate) \Rightarrow G\ specializes\ SATEL.HMLFormula.InputEvent)$
- $(\forall_M\ M \in \mathcal{P}(SATEL.APN.Method) \Rightarrow M\ specializes\ SATEL.HMLFormula.OutputEvent)$

4.3 Preliminares da solução

Uma descrição geral da solução foi apresentada no capítulo 1, onde a figura 4.1 foi exposta e uma pequena descrição de cada uma das etapas que dela constam foi apre-

Listagem 4.3: Excerto do Metamodelo SATEL

```
class HMLNext extends HMLFormulaContent
{
    property hmlFormulaContent :
        HMLFormulaContent[1] { !ordered,
        composes };
    property hmlEvent : HMLEvent[1] { !
        ordered,composes };
}
class HMLEvent
{
    property inputTerm :
        SynchronizationInputTerm[1] { !
        ordered,composes };
    property outputTerm :
        SynchronizationOutputTerm[?] { !
        ordered,composes };
}
class SynchronizationTerm { abstract };
class SynchronizationInputTerm extends
    SynchronizationTerm { abstract };
class SynchronizationEventInputTerm extends
    SynchronizationInputTerm
{
    property event : InputEvent[1];
    property parameters : Parameter[?] {
        composes };
}
class SynchronizationOutputTerm extends
    SynchronizationTerm { abstract };
```

Listagem 4.4: Excerto do Metamodelo do SATEL

```
    class SynchronizationEventOutputTerm
        extends
        SynchronizationOutputTerm
    {
        property event : OutputEvent[1];
        property parameters : Parameter
            [?] { composes };
    }
    class WPrimitiveObservationVarDec
        extends
        SynchronizationOutputTerm
    {
        property primitiveObservation :
            VariableDeclarations::
            PrimitiveObservationVarDec
            [1];
    }
    class WPrimitiveStimulationVarDec
        extends SynchronizationInputTerm
    {
        property primitiveStimulation :
            VariableDeclarations::
            PrimitiveStimulationVarDec
            [1];
    }
```

sentada. Para uma melhor compreensão de cada uma delas é necessário conhecer o metamodelo que se encontra no apêndice A.1.

Como primeira etapa temos o pré-processamento dos modelos SATEL⊕EAPN que nos remete para o tratamento de variáveis e estruturas que necessitam de *unfolding*. Ao nível das variáveis é feita a alteração do nome da cada variável presente nos modelo SATEL⊕EAPN. Como necessitamos que os nomes das variáveis sejam maiúsculas (requisito do Prolog), visitamos todas as variáveis incrementando um índice inicializado a zero no início do processamento. O nome de cada variável será uma letra maiúscula concatenada com esse índice. As entidades que representam variáveis que são alteradas são os VariableDec, que são as variáveis do SATEL, e os Variable referentes às variáveis das APN e, mais particularmente, aos ADT que lhe estão associados).

Nesta fase preliminar é feito também o *unfolding* dos termos algébricos complexos para que a transformação na ferramenta DSLTrans se torne possível. Esta alteração na realidade não altera o significado do modelo de entrada.

Recordemos a notação $suc^{10}(zero)$ que significa aplicação sucessiva (dez vezes) da operação *suc*. Em termos de sintaxe concreta do editor implementado, a notação é idêntica, e em termos de metamodelo a representação é, também, similar (listagens 4.6 e 4.5 correspondentes aos elementos CompositeTerm das AlgebraicExpressions e CTerm

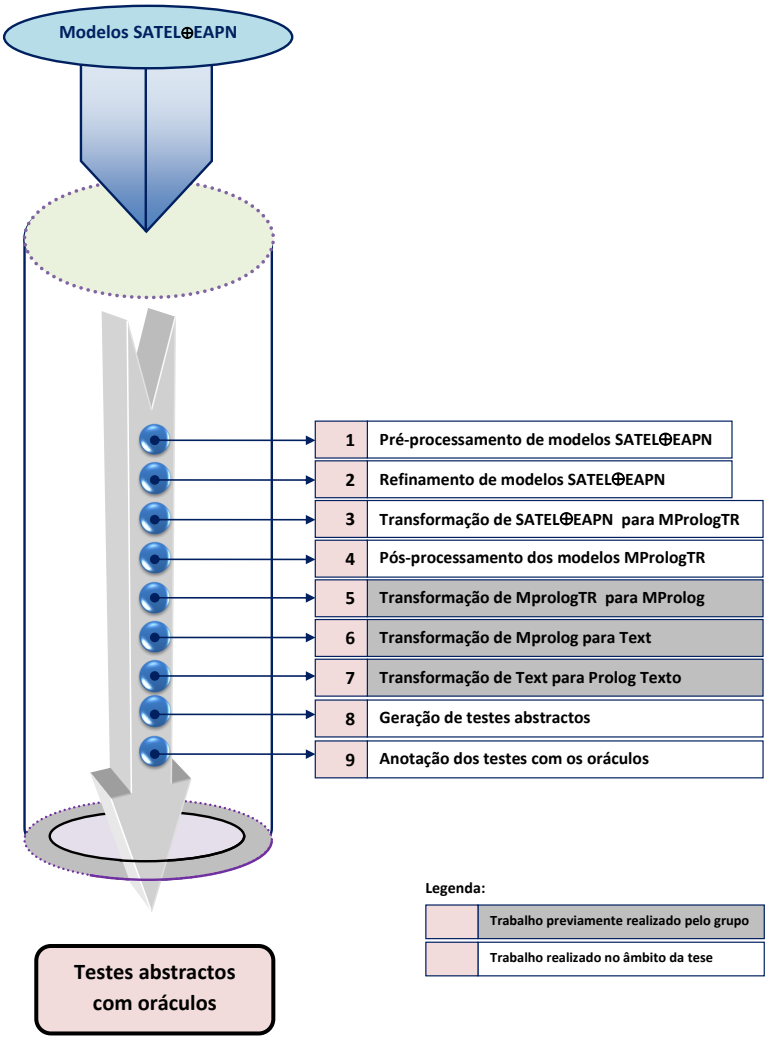

Figura 4.1: Visão geral da solução proposta

dos ADT das APN, respectivamente).

Listagem 4.5: Metamodelo SATEL - CTerm

```
1 class CTerm extends Term
2             {
3                 attribute iter : ecore_0::
                      EInt[?];
4                 property ownedTerms : Term
                      [*] { composes };
5                 property op : Operation[1];
6             }
```

Listagem 4.6: Metamodelo SATEL - CompositeTerm

```
1 class AbstractCompositeTerm extends
      AlgebraicTerm { abstract }
2 {
3     property terms : AlgebraicTerm[*] { !
          ordered,composes };
4     property op : adtmm::Operation[1] { !
          ordered,!unique };
5     attribute iter : ecore_0::EInt[?] = '0';
6 }
7 class CompositeTerm extends
      AbstractCompositeTerm;
8 }
```

Se se reparar, ambos têm um atributo iter e é sobre esse atributo que vamos fazer *unfolding*. Assim, como exemplo, teremos:

$$suc^3(zero) = CTerm(op, 3, [zero]) = CTerm(op, 1, CTerm(op, 1, CTerm(op, 1, [zero])$$

Após ter sido feito o pré-processamento, é feito o refinamento do modelo. Este refinamento encontra-se no apêndice A.4, e serve para facilitar a transformação principal com a ferramenta DSLtrans. Com efeito a *Eclipse Modeling Framework* representa listas em formato XML (XMI), mas a partir da ferramenta DSLTrans não é possível obter iterativamente cada um dos elementos de uma lista pela ordem que estão colocados, deste modo é necessário carregar o modelo com alguma redundância colocando em cada objecto de uma lista, um apontador para o seguinte nessa lista.

Por esse motivo foi adicionado um apontador next na declaração do tipo dos objectos pertencentes a listas, que necessitem ser processadas como lista e não como conjuntos, estendendo o metamodelo SATEL⊕EAPN. Desta forma, manteve-se a compatibilidade com o AlPiNa. Este refinamento foi implementado através de ATL em *refining mode*, linguagem de transformação baseada em modelos apresentada em trabalho relacionado, que se recorre de OCL estendido e que permite construções iterativas.

A listagem 4.7 contempla um excerto da transformação referida. Como se pode observar na linha 14 temos uma das várias regras presentes na transformação. Neste caso temos a regra para objectos ConditionBody que pode conter diversos ConditionAtom. Para cada um desses ConditionAtom é invocado um *helper* que devolve o elemento seguinte na lista de ConditionAtom e cujo valor é afectado na propriedade next. Note-se ainda que a transformação é endógena e executada em modo de refinamento (ver linha 2 - '*refining*').

Listagem 4.7: Excerto da transformação em ATL

```
module refineSATELAPN;
create OUT : SATELAPN refining IN : SATELAPN;

helper def : setNext(seq: Sequence(OclAny), elt: OclAny) : OclAny=

    elt.refSetValue('next',
                    if elt<> seq.last() then
                                    seq.at(seq.indexOf(elt)+1)
                                else
                                   OclUndefined
                            endif.debug()
 )
;
rule ConditionBody {
    from
        condBody : SATELAPN!ConditionBody
    to
        out : SATELAPN!ConditionBody (
                conditionAtom <- condBody.conditionAtom->asSequence()->
                        collect(e | thisModule.setNext(condBody.conditionAtom, e))
        )
}
}
```

4.4 Metamodelação do Prolog

Utilizar Prolog para obter os testes acabou por se tornar, por fim, a solução ideal porque é uma linguagem declarativa que se assemelha às notações usadas nos modelos de teste e que por isso permite expressar de forma muito directa a semântica do SATEL e das intenções de teste.

A abordagem proposta consiste em transformar o modelo especificado em SATEL⊕EAPN em cláusulas Prolog. Para isto efectuou-se uma transformação de SATEL⊕EAPN para MPrologTR (MPrologTermReference), cujo metamodelo se encontra na figura 4.2. A utilização de um Prolog Metamodelado faz todo o sentido nesta dissertação, tendo em conta a natureza, tanto do problema como da solução que se baseia em modelos. Este metamodelo do Prolog ainda que simplista e muito ligeiramente relaxado permite, geralmente, expressar as construções da linguagem Prolog, e em particular, tem a expressividade necessária para desenvolver este trabalho e para poder ser reutilizado noutros trabalhos e abordagens que dele necessitem.

De uma maneira detalhada, o metamodelo MPrologTR relaciona-se com a linguagem Prolog da forma que se apresenta na tabela 4.1

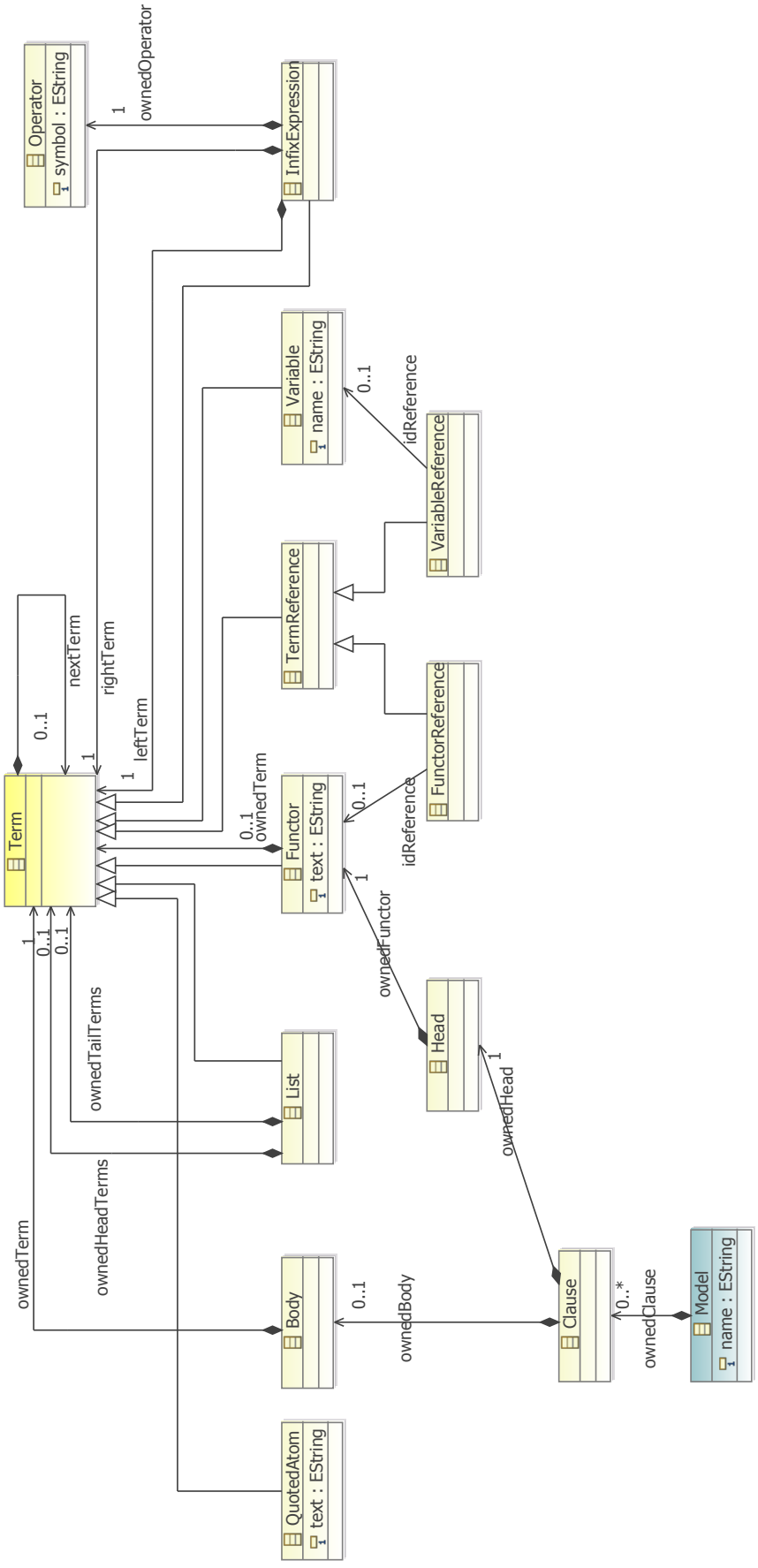

Figura 4.2: Metamodelo do MPrologTermReference

Conceito Prolog	Classe em MPrologTR	Exemplo
Cláusula	Clause	c(X):-b(X).
Corpo de Cláusula	Body	c(X):- b(X), a(X).
Cabeça de Cláusula	Head	c(X):-b(X).
Lista	List	c([X | Y]):-b(X).
Functor	Functor	a(X)
Operador Infixo	InfixExpression	L =.. X
Variável	Variable	c(X)
String	Quoted Atom	'atom'
Termo seguido de Termo	Term.next (Cada Termo tem uma referencia para o próximo termo)	:-c(X), b(X) **

Tabela 4.1: Relação entre MPrologTR e conceitos da linguagem Prolog

De notar ainda que List, Functor, Variable, InfixExpression e Variable são, por herança, um Term, e que Head tem um ownedFunctor:Functor, que por restrição OCL tem *next* indefinido, e Body tem um OwnedTerm:Term que será o primeiro dos termos do corpo da cláusula. Os outros são obtidos pela desreferenciação sucessiva de *next*.

Note-se ainda os elementos FunctorReference e VariableReference que permitem apontar para um Functor ou Variable já existentes e contidos noutro elemento. Estes elementos são resolvidos por cópia de nome aquando da transformação de MPrologTR para MProlog, pois o MProlog diferencia-se do primeiro por não possuir referências para Term.

4.5 Semântica operacional da SATEL

No presente capítulo apresenta-se uma listagem das cláusulas Prolog que implementam a semântica operacional da SATEL. Na prática para cada ConditionAtom ou ArithmeticTerm previsto na linguagem SATEL há uma cláusula que será utilizada para avaliar esse ConditionAtom ou ArithmeticTerm aquando da geração dos testes. A estrutura utilizada segue um formato em que existe um functor *eval* à cabeça de aridade dois e que devolve o resultado no segundo argumento. O primeiro argumento é o termo a ser avaliado.

```
1  %ConditionAtom
2  eval(boolNot(B), R ):-eval(B,BR), b_not(BR,R).
3  eval(boolAnd(L,R), Rs):- eval(L,LL), eval(R,RR), b_and(LL,RR,Rs).
4  eval(boolOr(L,R), Rs):- eval(L,LL), eval(R,RR), b_or(LL,RR,Rs).
5  eval(boolEquals(L,R), Rs):- eval(L,LL), eval(R,RR), b_equals(LL,RR,Rs).
6  eval(boolNotEquals(L,R), Rs):- eval(L,LL), eval(R,RR), b_equals(LL,RR,RA), b_not(RA,Rs).
7  eval(boolGT(L,R), true):-eval(L,LL), eval(R,RR), LL>RR,!.
8  eval(boolGT(L,R), false).
9  eval(boolLT(Z,Y),true):-eval(Z,ZZ), eval(Y,YY), ZZ < YY.
10 eval(boolGTE(L,R), true):-eval(L,LL), eval(R,RR),LL>=RR,!.
11 eval(boolGTE(L,R), false).
12 eval(boolLTE(L,R), true):-eval(L,LL), eval(R,RR), LL=<RR,!.
13 eval(boolLTE(L,R), false).
14 eval(boolSequence([hmlAnd(L,R)| X]), false):-!.
```

```
15 eval(boolSequence([hmlNext(H,K)|T]),R):-eval(boolSequence(T),X), eval(boolSequence(K),X1),
       b_and(X,X1,R),!.
16 eval(boolSequence([Z|T]), R):-eval(boolSequence(T,R).
17 eval(boolPositive([hmlAnd(L,R)| X]), false):-!.
18 eval(boolPositive([hmlNext(H,K)|T]),R):-eval(boolSequence(T,X), eval(boolSequence(K,X1), b_and
       )X,X1,R),!.
19 eval(boolPositive([Z|T]), R):-eval(boolPositive(T),R).
20 eval(boolTrace(T), R):- eval(boolPositive(T),true), eval(boolSequence(T),true).
21 eval(hmlEquality(L,R),true):-eval(L, LL), eval(R, RR), LL=RR.
22 eval(hmlEquality(L,R),false):-eval(L,LL), eval(R,RR),LL\=RR.
23 eval(algEquality(L,R),true):-eval(L, LL), eval(R, RR), LL=RR.
24 eval(algEquality(L,R),false):-eval(L,LL), eval(R,RR),LL\=RR.
25 eval(booleanEquality(L,R),true):-eval(L, LL), eval(R, RR), LL=RR.
26 eval(booleanEquality(L,R),false):-eval(L,LL), eval(R,RR),LL\=RR.
27
28 %ArithmeticTerm
29 eval(bopPlus(L,R), R):-eval(L,LL), eval(R,RR), R is LL+RR.
30 eval(bopMinus(L,R), R):-eval(L,LL), eval(R,RR), R is LL-RR.
31 eval(bopDiv(L,R), R):-eval(L,LL), eval(R,RR), R is LL/RR.
32 eval(bopTimes(L,R), R):-eval(L,LL), eval(R,RR), R is LL*RR.
33 eval(uopMinus(V), R):-eval(V,VV), R is -1*VV.
34 eval(arithmeticEquality(L,R),Res):-eval(L,LL), eval(R,RR), b_equals(LL,RR, Res).
35 eval(nbEvents([]),0):-!.
36 eval(nbEvents([top]),0):-!.
37 eval(nbEvents([top|T]),R):-eval(nbEvents(T),R),!.
38 eval(nbEvents([(_,_)|T]),R):-eval(nbEvents(T),R1), R is R1+1.
39 eval(depth([]),0):-!.
40 eval(depth([top]),0):-!.
41 eval(depth([top|T]),R):-eval(depth(T),R),!.
42 eval(depth([(_,_)|T]),R):-eval(depth(T),R1), R is R1+1.
43
44 eval(in(X,H),true):-in(X,H).
45 %Tabela de verdade de operadores auxiliares
46 b_not(true,false).
47 b_not(false, true).
48 b_and(true, true, true):-!.
49 b_and(X,Y,false).
50 b_or(false,false,false):-!.
51 b_or(X,Y,true).
52 b_equals(X,X,true).
53 b_equals(X,Y,false):-X\=Y.
54
55 eval(X,X):-atom(X).
56 eval(Y,Y):-number(Y).
```

Da implementação da semântica operacional do SATEL há a referir que se obteve algo muito similar à especificação formal o que potencia uma melhor análise a desvios de comportamento que possam a existir aquando da execução do código Prolog resultante.

4.6 Geração de Oráculos

A obtenção dos oráculos é feita partindo das listas de estímulos finais representados em listas Prolog, aplicando cada estímulo à semântica operacional da EAPN, observando se é possível através desse estímulo, as transições associadas dispararem e se subsequentemente são observados os *outputs* esperados.

Tome-se como exemplo o traço de execução(teste) da listagem 4.8 que foi gerado pela ferramenta.

Listagem 4.8: 'Teste gerado'

```
1 [ (tick, null), (tick, null), (tick,null), (mark with time(suc(suc(suc(zero))))) ]
```

Este traço de execução do sistema corresponde à invocação de tick por três vezes finalizando com mark observando-se na *gate* time o valor suc(suc(suc(zero))) possivelmente correspondente ao número de ticks de *input*. Este teste, em rigor, deverá passar porque o número de ticks está de acordo com a observação em time. Para que seja calculado o oráculo é necessário que a APN seja transformada em Prolog da forma que se explicará em seguida. Em particular, as regras de transformação serão explanadas no próximo capítulo, no entanto como é necessário compreender que tipo de código Prolog será gerado a partir do modelo para se poder introduzir o método de cálculo do oráculo, ir-se-á introduzir nesta secção o tipo de transformação que deve ser feita.

A EAPN de referência nesta dissertação contém dois *places*. Um dos *places* está inicialmente com o *token* algébrico zero e o *place* correspondente ao estado final não contém quaisquer *tokens*. A partir destes factos modelados é produzida a linha Prolog da listagem 4.9 que mais não é do que uma lista com um *functor place* com dois argumentos: o nome da transição e uma lista dos *tokens* algébricos que se encontram inicialmente na marcação.

Listagem 4.9: 'Marcação inicial em Prolog'

```
1 [ place( 'Inc',[ zero ] ) , place('Finished',[]) ]
```

De seguida transforma-se as transições da seguinte forma: por cada transição que conste da EAPN gera-se uma cláusula como indicado na listagem 4.10.

Listagem 4.10: 'A transição em Prolog'

```
1 transition(STIMULUS, OBSERVATION, MARCATION, NEWMARCATION):-
2                         takeout(OUTTOKENS, PLACEA,MARCATION, NEW_MARCATION_INTERM),
3                                                 putinMany(INTOKENS, PLACEB,
                                                      NEW_MARCATION_INTERM, NEWMARCATION),
4                                                 <Conditions> .
```

Como se pode observar, a cláusula tem como cabeça o functor 'transition' de aridade 4, onde recebe como argumento o método de *input* que despotela a transição, a

gate de observação, a marcação que a EAPN apresenta num determinado momento e a marcação nova caso a cláusula suceda.

No corpo da cláusula tem-se o predicado takeout que recebe como argumento uma lista de *tokens* que são consumidos pela transição, o *place* de onde deve ser consumido; a marcação que a EAPN apresenta e a nova marcação após o consumo dos *tokens*. Tem ainda o predicado putinMany que recebe uma lista com os *tokens* que devem ser depositados no *place* PLACEB e a marcação que resulta do predicado takeout e que devolve a marcação existente após a colocação dos *tokens* no *place* de destino. Por fim devem ser acrescentadas ao corpo da cláusula todas as condições algébricas.

O predicado takeoutMany funciona em cooperação com o predicado takeout tal como se observa na listagem 4.11, e permite retirar vários *tokens* do *multiset* de um *place*. Deste modo, o que acontece neste predicado é um processamento elemento a elemento da lista de *tokens* a retirar. Por cada elemento é utilizado o predicado takeout (linha 4) para fazer a eliminação do valor algébrico em processamento (AlgebraicValue) do *multiset*do *place* com nome 'Name', numa dada marcação. Após o processamento desse valor algébrico passa-se ao seguinte na lista com a recursão sobre takeoutMany com o resto da lista (Tail) e onde a marcação de entrada é a marcação de saída após a eliminação do elemento.

O predicado takeout, por sua vez, procura através do predicado member, um *place* que esteja na marcação que tenha o nome dado como *input*; a partir daí extrai-se o *MultiSet* que lhe está associado (linha 8). Na linha seguinte é utilizado o predicado auxiliar deletefrom para efectivamente retirar o *place* da marcação e na linha 10 é, através do predicado member, procurado um valor no *Multiset* do *Place* que unifique com AlgebraicValue. Por fim, na linha seguinte elimina-se o valor do *multiset*. A marcação de retorno é construída na primeira linha do predicado, onde lhe é adicionado o *place* retirado já com o novo *multiset* (sem os *tokens*).

Nesta solução entra-se em linha de conta com a característica do predicado member, cujos argumentos são de entrada e saída simultaneamente. Desta forma se o *multiset* do *place*em questão contiver vários elementos (e.g. [a,b,c]), e se se pretender retirar vários também, o Prolog procura por *backtracking* uma combinação de unificação entre os elementos a retirar e os elementos do *place*.

Mais se acrescenta que foi concretizada uma solução com o predicado select[1], mas para abono da compreensão e para tornar a implementação mais próxima daquilo que realmente acontece com a EAPN em termos semânticos, optou-se por apresentar esta.

Listagem 4.11: 'Predicados takeout e takeoutMany'

```
1 takeoutMany([],_,Marcation,Marcation).
```

[1]http://www.swi-prolog.org/pldoc/docfor?object=select/3

```
2
3  takeoutMany([AlgebraicValue|Tail], place(Name), Marcation, NewMarcation):-
4                  takeout(AlgebraicValue, place(Name), Marcation, NewIntermMarcation),
5                              takeoutMany(Tail, place(Name), NewIntermMarcation, NewMarcation).
6
7  takeout(AlgebraicValue, place(Name), Marcation, [place(Name,NewMultiSet) | NewMarcation]):-
8                                                                  member(place(Name,MultiSet),
                                                                       Marcation),
9                                                                  deletefrom(place(Name,MultiSet)
                                                                       ,Marcation,NewMarcation),
10                                                                 member(AlgebraicValue,MultiSet)
                                                                       ,
11                                                                 deletefrom(AlgebraicValue,
                                                                       MultiSet,NewMultiSet).
12
13 deletefrom(_,[],[]).
14 deletefrom(P, [P|T],T):-!.
15 deletefrom(X, [P|T],[P|TailResult]):-deletefrom(X,T,TailResult).
```

Em oposição ao funcionamento do predicado takeoutMany tem-se o predicado putinMany que permite colocar elementos no *multiset* de um *place*. O modo de funcionamento do predicado putinMany é similar ao do takeoutMany exceptuando-se o facto de que após ser encontrado o *place* pretendido é feita uma concatenação com o seu *multiset* com o fim de lhe adicionar os elementos.

Listagem 4.12: 'Predicados putin e putinMany'

```
1  putinMany(AlgebraicValues, place(Name), Marcation, [place(Name, NewMultiSet)| NewMarcation]):-
2                                                  member(place(Name,MultiSet), Marcation),
3                                                  deletefrom(place(Name,MultiSet), Marcation
                                                       , NewMarcation),
4                                                  append(AlgebraicValues,MultiSet,
                                                       NewMultiSet).
```

Conclui-se esta explanação com um exemplo concreto de transformação da transição que retira um *token* do *place* 'Inc' e lhe coloca o seu sucessor. Repare-se na linha 1 da figura 4.13, em que o método de input é tick e não tem observação. Na linha 2 pretende-se que seja retirado o *token* Counter do *place* 'Inc' que, em rigor, unificará com $zero$ aquando do primeiro disparo. Na linha 3 instrui-se para que seja colocado de volta o sucessor do *token* retirado. Por fim, encontra-se a condição da transição obrigando a que Counter não exceda $suc^{10}(zero)$.

Listagem 4.13: 'Exemplo de transformação de uma transição'

```
1  transition(tick, null, Marcation, NewMarcation):-
2              takeout(Counter, place('Inc'),Marcation, NewMarcation1),
3              putin(suc(Counter), place('Inc'),NewMarcation1, NewMarcation),
4              eval(algEq(lessThan( Counter,suc(suc(suc(suc(suc(suc(suc(suc(suc(zero)))))))))
                   )), true), true).
```

O oráculo é, por fim, calculado através do predicado da listagem 4.14. Tem-se um caso base em que se a lista que contém o traço de execução apenas contiver o ele-

mento Top, o oráculo retornado é verdadeiro. Tem-se ainda na linha 2, o caso geral; a cabeça da lista do traço de execução é unificada com o par input(I)/output(O), e o predicado apenas sucede se existir uma transição que seja despoletada com o par I/O com a marcação que é dada como argumento; existindo, a execução avança com a marcação retornada e com o próximo elemento da lista do padrão de execução. Chegando ao caso base o predicado sucede com um oráculo verdadeiro. Se em algum momento não houver uma transição que possa ser despoletada cai-se no caso da terceira linha e oráculo que lhe fica associado é falso.

Listagem 4.14: 'Exemplo de transformação de uma transição'

```
1 oracle(([top], true),_):-!.
2 oracle(([(I,O)|Tail], true),Marcation):-transition(I,O,Marcation, NewMarcation), oracle((Tail,
      true),NewMarcation),!.
3 oracle((_,false),_).
```

Pode-se assim concluir que a geração do oráculo é inerente à semântica operacional da EAPN. Em rigor, a transformação da EAPN é feita levando em conta que para o cálculo do oráculo é necessário que existam cláusulas adequadas para esse efeito. Assim as cláusulas referentes à semântica operacional da EAPN remetem para o cálculo do oráculo.

4.7 Modelo de transformação

Neste capítulo pretende-se mostrar como é levada a cabo a transformação de um modelo SATEL⊕EAPN para um modelo MPrologTR. Devido à dimensão do modelo de transformação, este encontra-se no apêndice A.8 na sintaxe textual desenvolvida no âmbito desta dissertação, sendo que nesta secção iremos introduzir um pequeno exemplo para familiarizar o leitor com a notação e apresentaremos o mapeamento pretendido entre a sintaxe concreta do SATEL⊕EAPN e a linguagem Prolog para melhor explicitar o objectivo final da transformação.

4.7.1 Regras de transformação

Para transformar o modelo foram especificadas um número avultado de regras que cooperativamente permitem obter o código Prolog final. Nesta secção apresentar-se-á, de forma intuitiva, as regras envolvidas na transformação de diversos trechos de um modelo especificado através da sintaxe textual do SATEL⊕EAPN. Para que a descrição não se fique pela forma intuitiva (logo informal), à medida que forem descritas as regras indicar-se-ão apontadores para o apêndice das regras em análise, especificadas na sintaxe do DSLTrans.

Posto isto, atente-se no exemplo da figura 4.3. Na transformação aqui envolvida, note-se que é um gerador de naturais cujo nome da operação associada é `zero` e cujo sort do resultado é claramente `nat` (naturais). Da figura depreende-se então que, em primeira análise, num gerador, dito constante, cria-se um predicado cujo nome é o sort do result, cuja aridade é um e o único argumento é um functor sem argumentos cujo nome é o nome do gerador.

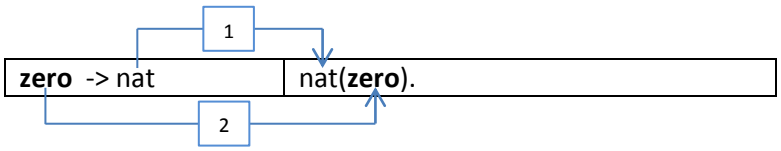

Figura 4.3: Transformação do gerador

De uma forma mais geral, atente-se na figura 4.4. Neste exemplo tem-se um gerador que possui argumentos de determinado sort. Repare-se que a regra a aplicar é similar à anterior com as devidas extensões/generalizações. Com efeito, aqui o sort do resultado (nat) volta a ser o nome do functor da cabeça de uma cláusula. Neste exemplo é necessário uma cláusula porque é necessário impor condições sobre os argumentos.

O functor da cabeça volta a imbricar o functor com o nome do gerador, no entanto, este agora tem aridade superior a zero. Os argumentos deste functor devem ser tantas variáveis quantos os argumentos do gerador, na mesma ordem de correspondência. No corpo da cláusula é necessário impôr que as variáveis criadas por via de cada um dos argumentos do gerador, sejam exactamente do sort do argumento correspondente (ex. fluxo 3 e 5) da figura.

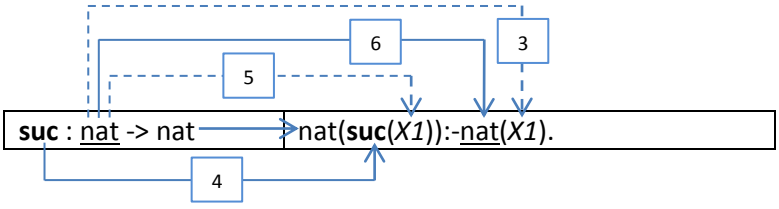

Figura 4.4: Transformação do gerador

Da mesma forma e já generalizando, na figura 4.5, ao invés de se ter apenas um argumento, tem-se dois; um para o elemento a adicionar à lista e outro para a lista onde vai ser adicionado à cabeça. Assim pode-se concluir que se deve criar um predicado com aridade um quando o gerador é constante e uma cláusula quando o gerador tem

argumentos. O nome do functor da cabeça é, nesse caso, o nome do sort do resultado, e contém um functor com tantas variáveis quantas os argumentos do gerador; variáveis essas que devem ser restringidas quanto ao seu sort no corpo da cláusula.

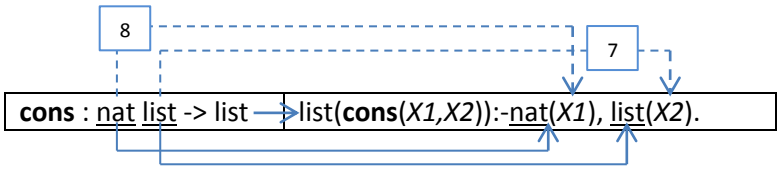

Figura 4.5: Transformação do gerador

Para a transformação dos geradores são assim necessárias regras para as seguintes tarefas:

- criação de um objecto Clause por cada gerador
- criação de um objecto Head para a Clause
- criação de um objecto Body para a Clause
- criação do functor com o nome do gerador (A)
- criação do functor com o nome do sort do resultado (B)
- criação de variáveis por cada argumento do gerador, para colocar em Head (V)
- criação de functores por cada argumento do gerador, para colocar em Body (FV)
- relacionar Clause com Head
- relacionar Clause com Body
- relacionar Functores (A) e (B)
- relacionar Functores (FV) com as Variáveis respectivas (V)
- relacionar object Clause com o objecto Model

Tendo já sido definidas as regras para os geradores de um ADT, observe-se a figura 4.6. A formulação empírica presente na regra corresponde à transformação de um axioma sobre uma operação de um ADT.

Este é um axioma sobre a operação nbOcurr que pode ser aplicado quando se verifica a condição a azul. Em cima, a laranja, tem-se o lado esquerdo da regra e a vermelho a reescrita dos termos que estão à esquerda.

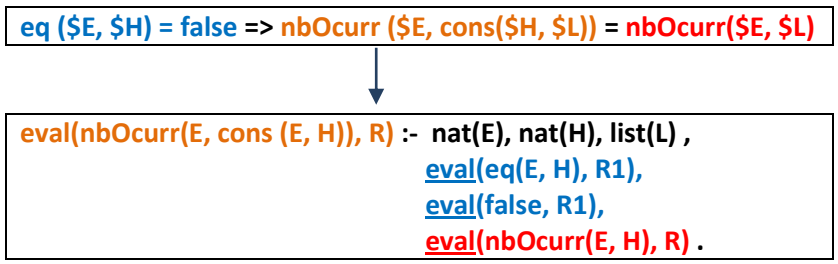

Figura 4.6: Transformação dos axiomas das operações

Posto isto, na transformação de um axioma deve ser criada uma cláusula. Nesta cláusula, a cabeça deve ser um functor com nome 'eval' que tem como primeiro argumento aquilo que se pretende reescrever e como segundo argumento uma variável que conterá o resultado da avaliação/redução do primeiro. Em particular, doravante, os functores 'eval' funcionam sempre desta forma.

O corpo da cláusula deve conter nesta ordem, os functores que restringem o sort dos elementos envolvidos na operação seguidos de functores 'eval' para ambos os membros das equações das condições do axioma. Neste caso adiciona-se um functor 'eval' que deve avaliar eq(E,H) e colocar o resultado numa variável (R1) e coloca-se outro functor que avalia false e tenta unificar R1 para ambos os predicados. Por fim coloca-se um functor eval que avalia o lado direito da equação do axioma e que unificará o resultado com R (que é o resultado do lado esquerdo). Se houvesse mais condições, os respectivos predicados deveriam ser postos antes deste functor para que as condições fossem todas avaliadas previamente.

A esta estratégia há pelo menos duas adições a fazer. Em caso de condições que sejam inequações é necessário que para ambos os resultados (lado esquerdo e direito) sejam criadas variáveis diferentes, e garantir que elas não unificam (e.g. R1\=R2, em que R2 é a variável correspondente ao lado direito e R1 ao lado esquerdo).

A segunda adição tem que ver com o facto de os geradores terem que poder ser avaliados. Por exemplo em eval(false,R1), onde consta o gerador false dos booleanos, quer-se que R1 unifique com false, mas se um fosse um natural (e.g. eval(suc(X),R2)) querer-se-ia que R2 unificasse com suc(X), em que X já estivesse avaliado. Foi para este efeito que se criaram regras que resolvem os exemplos das figuras 4.7 e 4.8, as quais serão explicadas em seguida.

Posto isto foram criadas regras para o seguinte:

- Criação de Clause para cada CondEquation (Axioma)
- Criação de Body e Head para cada CondEquation

- Criação de functores eval para cada lado de uma equação ou inequação
- Criação de variáveis para as variáveis envolvidas num axioma
- Criação de variáveis para os diversos resultados
- Criação de functores que representem os CTerm (e.g. suc(suc(zero)))
- Criação das relações nextTerm entre ConditionAtom (Condições)
- Criação de functores para restringir o sort de cada variável
- Criação de functores 'eval' para os geradores tal como indicado nas figuras 4.7 e 4.8.

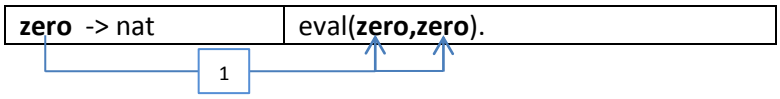

Figura 4.7: Transformação do gerador

suc :nat -> nat	eval(suc(X1),suc(XX1)):-eval(X1,XX1).

Figura 4.8: Transformação do gerador

Por fim, tem-se regras para transformar a especificação das intenções de teste. O código Prolog pretendido para a transformação dos axiomas das intenções de teste é muito similar ao código pretendido para os axiomas dos ADT. Exceptua-se apenas pelo facto de as intenções de teste poderem dar origem a cláusulas sem corpo, como no caso da figura 4.9.

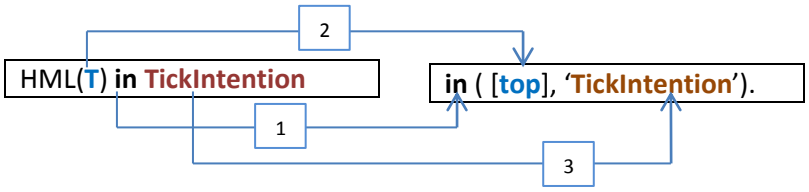

Figura 4.9: Transformação das intenções de teste

Em relação aos axiomas das intenções de teste quer-se que estes resultem numa cláusula cuja cabeça é um functor 'in' (fluxo 1 da figura 4.9), com aridade 2, cujos argumentos são primeiramente uma lista com o padrão de execução do sistema (fluxo

2), e o segundo é um *quoted atom* com o nome da intenção (fluxo 3). Caso existam condições os correspondentes functores 'eval' devem ser colocados no corpo da cláusula (e.g. condições a azul e a verde na figura 4.10).

O padrão de execução pode conter o objecto HMLTop (que se representará com um functor top sem argumentos) e eventos que devem ser representados por um par ordenado em que o primeiro elemento é um functor correspondente ao método de *input* e o segundo é um functor correspondente à *gate* de *output*; ambos podem conter argumentos correspondentes aos do método e da *gate*, respectivamente (e.g. a sublinhado na figura 4.10).

```
{lt($Counter, suc^3(zero))} = {t r u e} , $t in TickIntention
                    =>
                              $t . HML( { <mark with time($Counter ) > } T ) in MarkIntention ;
```

```
1 in([T,(mark,time(Counter)),top], 'MarkIntention'):-
2
3   in(F,'TickIntention'),
4   flatten(F,T),
5   eval(algEquality(lt(Counter,suc(suc(suc(zero)))), true), true).
```

Figura 4.10: Transformação das intenções de teste

Assim se conclui, que em relação à transformação das intenções de teste é necessário criar regras para o seguinte:

- Criação do functor 'in' para cada axioma
- Criação da lista que contém o padrão de execução
- Criação de um 'QuotedAtom' referente ao nome da intenção
- Criação de functor eval para ConditionAtom (Condições)
- A cada Inclusion que seja ConditionAtom deve estar associado o predicado flatten
- O necessário para transformar CompositeTerm (e.g. suc(suc(zero))). É equivalente aos CTerm no contexto das intenções de teste
- O necessário para construir todo o ConditionAtom
- Criação de todas as relações necessárias para associar os objectos

4.7.2 Exemplificação das regras de transformação

Nesta secção pretende-se mostrar algumas das regras que foram especificadas para efectuar a transformação de modelos SATEL⊕EAPN em modelos MPrologTR. Não sendo frutífero ou vantajoso mostrar exaustivamente todas as regras (pela dimensão da transformação), apresentar-se-ão as regras que permitem obter as cláusulas correspondentes aos geradores dos ADT. Assim, por exemplo, para o ADT dos números naturais, cujos geradores estão apresentados na listagem 4.15, estão envolvidas as regras apresentadas em seguida.

Listagem 4.15: Geradores para os números naturais

```
1 zero -> nat
2 suc:  nat -> nat
```

No que diz respeito à primeira *layer* da execução da transformação, nesta está envolvida a regra da figura 4.11 (listagem 4.16) que permite gerar o objecto Model conformante com o metamodelo MPrologTR a partir do objecto Model da linguagem SATEL⊕EAPN. Este é o modelo que reúne em ownedClauses todas as cláusulas necessárias à geração dos testes. Por esse motivo é mantido o atributo (na listagem self = 'Model', na figura 'ApplyAtribute' como átomo designado 'Model') que permitirá referenciar este modelo nas *layers* subsequentes.

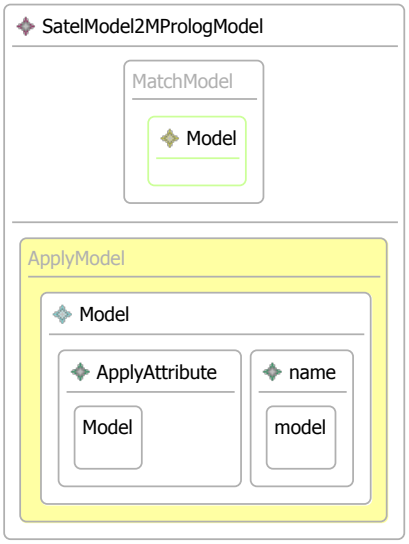

Figura 4.11: Regra: Transformação do modelo - Visual

```
1 rule 'SatelModel2MPrologModel'
2     match with
3         any SATEL::Model
4     apply
5         mprologTermReference::Model(
6             self = 'Model'
7             name = 'model'
8     )
9 end rule
```

Listagem 4.16: Regra: Transformação do modelo - Textual

Após a transformação do objecto Model é necessário, a partir de cada par de objectos ADT e Operation relacionados pela relação ownedGenerator, obter uma cláusula (i.e. um objecto Clause). Isto é efectuado através da regra 4.12 (listagem 4.17), onde uma vez mais é etiquetado o objecto gerado a partir deste par, com o valor 'GeneratorClause'.

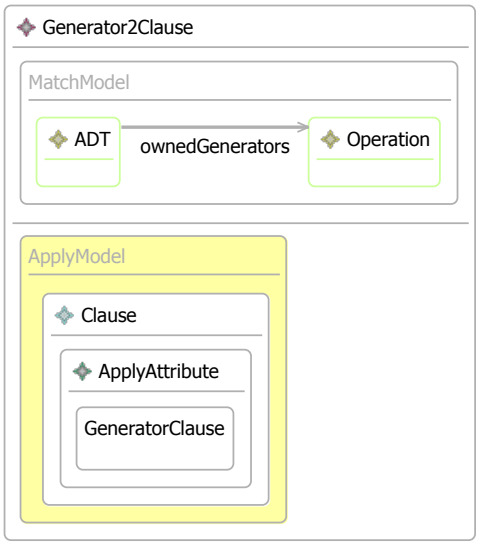

Figura 4.12: Regra: Cláusula do Gerador - Visual

```
1  rule 'Generator2Clause'
2      match with
3          adt: any SATEL.APN.adtmm::ADT
4          op: any SATEL.APN.adtmm::Operation
5
6          subject to
7                  adt --(ownedGenerators)-> op
8
9          apply
10             mprologTermReference::Clause(
11                 self = 'GeneratorClause'
12             )
13 end rule
```

Listagem 4.17: Regra: Cláusula do Gerador - Textual

No mesmo sentido, gera-se também um objecto Head e um objecto Body para cada par ADT e Operation relacionados pela mesma relação da regra anterior (figura 4.13 e figura 4.14 respectivamente).

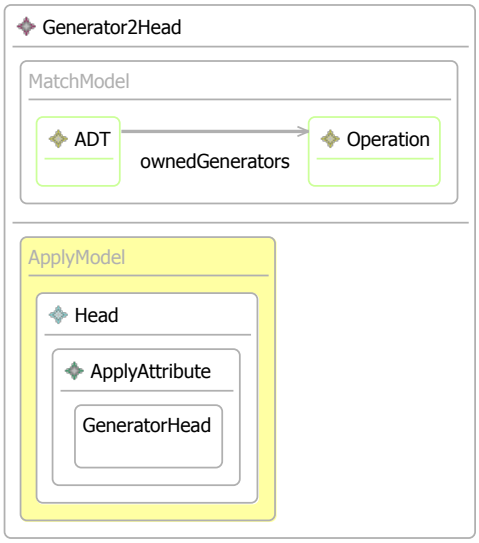

Figura 4.13: Regra: Cabeça do Gerador - Visual

```
1  rule 'Generator2Head'
2      match with
3          adt: any SATEL.APN.adtmm::ADT
4          op : any SATEL.APN.adtmm::Operation
5
6          subject to
7              adt --(ownedGenerators)-> op
8
9      apply
10         mprologTermReference::Head(
11                 self = 'GeneratorHead'
12             )
13 end rule
```

Listagem 4.18: Regra: Cabeça do Gerador - Textual

Em termos abstractos, com estas regras tem-se criada a estrutura de uma cláusula que representará um gerador (exceptuando-se as relações entre Clause e Body e entre Clause e Head).

Para dar seguimento à construção da cabeça da cláusula atente-se no exemplo do gerador suc: nat -> nat. Quer-se agora que nat seja o nome do functor da cabeça da

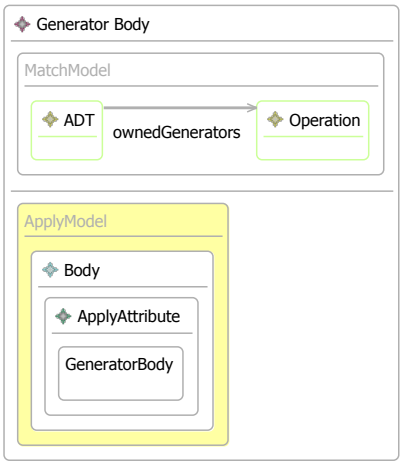

Figura 4.14: Regra: Corpo do Gerador - Visual

```
1  rule 'Generator Body'
2      match with
3          adt: any SATEL.APN.adtmm::ADT
4          op : any SATEL.APN.adtmm::Operation
5
6          subject to
7              adt --(ownedGenerators)-> op
8
9      apply
10         mprologTermReference::Body(
11                 self = 'GeneratorBody'
12         )
13 end rule
```

Listagem 4.19: Regra: Corpo do Gerador - Textual

cláusula para se obter algo como nat(...):- ... . Mais ainda, pretende-se que o functor nat imbrique outro functor, cujo nome é o nome da operação que constitui o gerador (suc), obtendo-se algo como nat(suc(...)):-... . Com a regra da figura 4.15 (listagem 4.20) identifica-se, tal como anteriormente, o padrão de gerador, conjuntamente com o atributo name da Operação e com o Sort do Resultado do gerador (e respectivo name em SortDeclaration). Tendo sido feita correspondência com o padrão geram-se os pretendidos functores: um, cujo nome (text) é o nome do respectivo sort do resultado e outro para ser imbricado no primeiro através da relação ownedTerm). Neste, o atributo text é o nome da operação.

Desta forma obtém-se em termos abstractos algo como nat(suc(...)):-... . Repare-se que ambos os functores são rotulados como 'ResultFunctor' e 'OperationFunctor', pois serão utilizados futuramente.

Para finalizar a definição das regras da primeira *layer* resta definir uma que gera os objectos necessários em falta para se obter nat(suc(X)):-nat(X). Daqui se depreende que serão criadas variáveis para cada um dos sorts (argumentos) do gerador para serem colocadas em ownedTerm do functor suc e serão gerados também functores por cada um dos sorts para serem colocados no corpo da cláusula. Na regra da figura 4.16 é novamente identificado o padrão de gerador conjugado com cada AtomicSort na lista de operationSorts (argumentos do gerador). Repare-se que o facto de ser uma lista faz com que este padrão seja identificado uma vez por cada elemento em operationSorts. Como se pode observar é gerado o functor com o nome do operationSort e é aninhado um objecto VariableReference através da relação ownedTerm. Este objecto VariableReference é um apontador para uma Variável (relação idReference) que será colocada no functor suc (self=OperationFunctor), criado na regra anterior. Repare-se que o nome

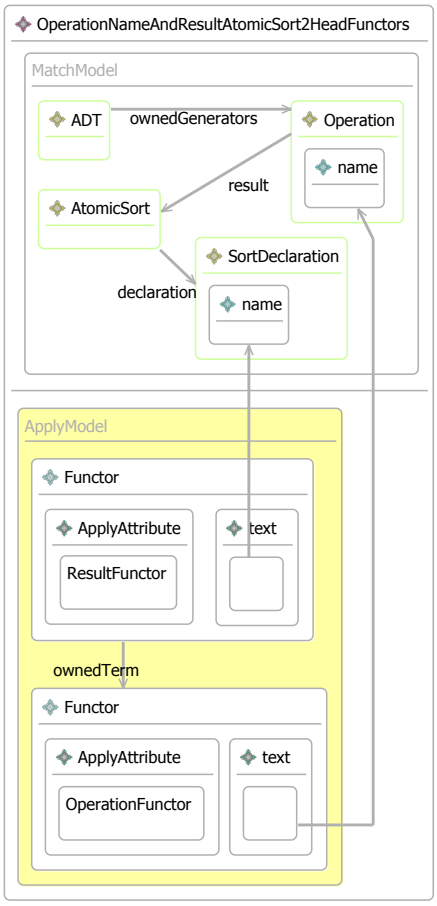

Figura 4.15: Regra: Constituintes da cabeça - Visual

```
1  rule 'OperationNameEResultSort2HeadFuncs'
2      match with
3       op: any SATEL.APN.adtmm::Operation(
4                                opname : name
5                    )
6       as: any SATEL.APN.adtmm::AtomicSort
7       sd : any SATEL.APN.adtmm::
             SortDeclaration(
8                                sdname : name
9                    )
10      adt: any SATEL.APN.adtmm::ADT
11
12      subject to
13       op  --(result)-> as
14         as  --(declaration)-> sd
15         adt --(ownedGenerators)-> op
16
17 apply
18      opFunc:
19          mprologTermReference::Functor(
20              self = 'OperationFunctor'
21              text= sameAs(opname)
22          )
23      resSortFunc:
24          mprologTermReference::Functor(
25                 self = 'ResultFunctor'
26                   text= sameAs(sdname)
27              )
28      subject to
29        resSortFunc --(ownedTerm)-> opFunc
30 end rule
```

Listagem 4.20: Regra: Constituintes da cabeça - Textual

da variável é '##var##', pois após o modelo ser criado é necessário percorrê-lo de forma a dar nomes apropriados a cada variável que tenha este padrão como nome. Isto deve-se a uma limitação do DSLTrans, uma vez que este peca por não conter, até à data, sequenciadores que permitam especificar situações como esta. Note-se ainda os rótulos dos objectos criados: OperationSortFunctor, AtomicSortVariableRef, AtomicSortVariable.

A partir deste ponto, todos os objectos necessários estão criados e passa-se à segunda *layer* onde serão criadas as relações entre estes.

Primeiro que tudo é necessário ligar a cláusula criada inicialmente ao modelo. A regra da figura 4.17 (listagem 4.21) identifica cada uma das operações que se encontram em algum nó das sub-árvores que derivam do nó Satel::Model e, por isso, utiliza-se uma conexão PositiveIndirectAssociation, isto é, uma associação indirecta. Na parte de *apply* da regra liga-se o Model, já criado na *layer* anterior, à Clause criada na *layer* anterior por via da mesma Operation. É para este efeito que são usados os PositiveBackwardLinks (a tracejado) que permitem forçar a que sejam utilizados exactamente

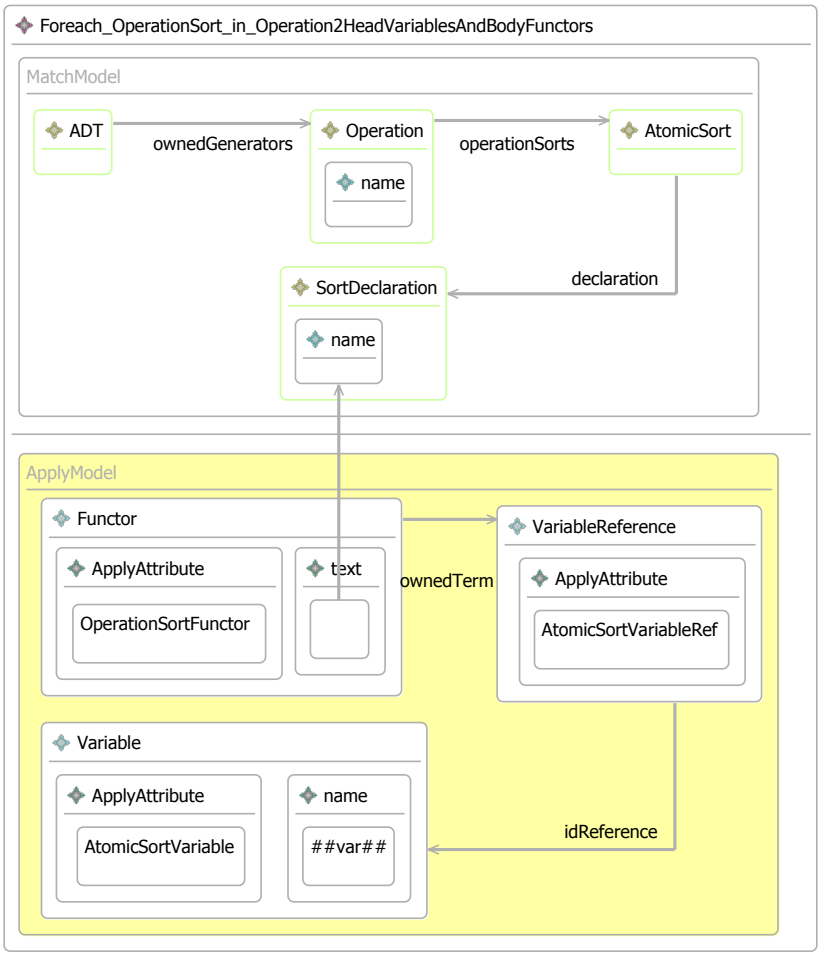

Figura 4.16: Regra: Variáveis à cabeça e functores do corpo da cláusula - Visual

os mesmo objectos criados na *layer* anterior a partir do mesmo objecto. Não obstante, é ainda necessário referir que o 'ApplyAttribute' tem o mesmo valor para que seja identificado o objecto. Neste caso 'GeneratorClause' para a cláusula gerada a partir da operação e 'Model' para o Model. Uma vez que há só um Model não seria necessário colocar o 'ApplyAttribute', contudo colocou-se para reforçar a ideia de que é o mesmo Model, e para que o leitor possa seguir melhor as regras seguintes.

Pode-se perguntar o que acontece quando é identificada uma Operation acessível a partir de Model e para a qual não existe uma Clause criada. Nesse caso a Operation identificada não é um gerador e, portanto não existe nenhum ADT que a contenha (via relação ownedGenerators). Consequentemente não existe o objecto Clause com ApplyAttribute 'GeneratorClause'. Neste caso a regra não se aplica simplesmente.

Da mesma forma associamos a cabeça (Head) da cláusula (Clause) à própria cláusula (através da relação ownedHead), tal como se mostra na figura 4.18(listagem 4.22). Repare-se que nesta regra, Head e Clause, tanto foram geradas a partir de ADT como de Operation, no entanto basta colocar uma restrição. A restrição em relação a Opera-

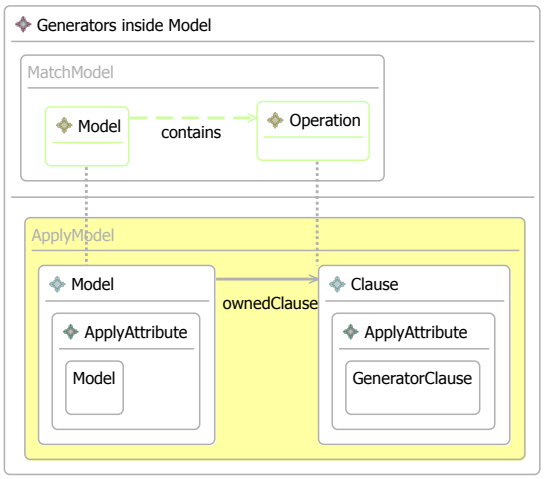

Figura 4.17: Regra: Cláusula no modelo - Visual

```
1  rule 'Generators inside Model'
2      match with
3          sm: any SATEL::Model
4          sop: any SATEL.APN.adtmm::
               Operation
5
6          subject to
7              sm ~~(contains)~> m261
8      apply
9          cl:
10                 mprologTermReference::
                       Clause(
11                     self = '
                           GeneratorClause'
12                 )
13         mmodel:
14                 mprologTermReference::
                       Model(
15                     ApplyAttribute= '
                           Model'
16                 )
17         subject to
18             mmodel --(ownedClause)-> cl
19
20     restrictions
21         cl derived from sop
22         mmodel derived from sm
23 end rule
```

Listagem 4.21: Regra: Cláusula no modelo - Textual

tion seria sempre necessária, pelo que permaneceu apenas essa, até porque a partir do momento em que o padrão está identificado sabe-se que a Operation é um gerador, e que, portanto, foi a partir desta gerado um objecto Head e um objecto Clause.

Na regra 4.19 (4.23), volta-se a identificar o Sort do resultado do gerador e recupera-se tanto a cabeça gerada a partir desse gerador como o functor ResultFunctor criado a partir da associação result entre Operation e AtomicSort na regra da figura 4.15.

Com a regra 4.20(listagem 4.24) recupera-se cada uma das variáveis associadas a cada um dos sorts que se relacionam com Operation através da relação operationSorts e que foram criadas nas regras da *layer* anterior. Repare-se que se utilizam exactamente os mesmos valores para os ApplyAttribute. Em rigor, colocamos em ownedTerm do functor ('OperationFunctor'), cada uma das variáveis. O laço no objecto Variable serve para quando um gerador tem mais do que um argumento (operationSort). Nesse caso esta regra é aplicada tantas vezes quantos os elementos em operationSort e, uma vez que o ownedTerm não é uma lista/conjunto e por isso apenas suporta um elemento, então ordena-se o DSLTrans para que, a seguir ao processamento do primeiro operationSort, a Variable seja colocada como seguinte (nextTerm) da variável que se encontra já lá colocada. Isto significa que o DSLTrans itera sobre nextTerm a partir do ownedTerm do functor até encontrar o fim desta lista.

Até este ponto, em relação ao gerador de exemplo tem-se: nat(suc(X)):-....

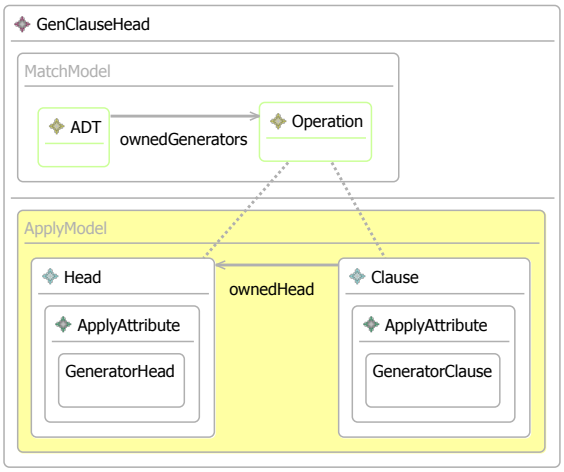

Figura 4.18: Regra: Cabeça na cláusula - Visual

```
1  rule 'Head inside Clause Gen'
2      match with
3          m264: any SATEL.APN.adtmm::
               Operation
4          m265: any SATEL.APN.adtmm::ADT
5          subject to
6              m265 --(ownedGenerators)->
                   m264
7      apply
8          m266: mprologTermReference::
               Clause(
9                      self = '
                           GeneratorClause'
10                 )
11         m267: mprologTermReference::Head
               (
12                     self = '
                           GeneratorHead'
13                 )
14         subject to
15             m266 --(ownedHead)-> m267
16     restrictions
17         m266 derived from m264
18         m267 derived from m264
19 end rule
```

Listagem 4.22: Regra: Cabeça na cláusula - Textual

Em seguida é necessário colocar os functores no corpo da cláusula. Com a regra da figura 4.21(listagem 4.25), à semelhança do que acontece com a regra anterior, faz-se a identificação das relações operationSort entre a Operation do gerador e os Sort e recupera-se o objecto Body que já lhe está associado e o Functor 'OperationSortFunctor' associado à relação. Mais uma vez é utilizado o laço sobre o functor para os colocar em nextTerm aquando de cada identificação de padrão.

Neste ponto tem-se o modelo da cláusula quase pronto, faltando apenas associar à cláusula o objecto Body gerado a partir da regra da *layer* anterior. A regra da figura 4.22 (listagem 4.27) funciona da seguinte forma: tenta-se identificar o padrão em que **exista** um objecto AtomicSort associado à Operation de um gerador através da relação operationSort. Caso a relação exista pelo menos uma vez então o objecto Body é colocado na Clause, caso contrário não é, porque o padrão da regra não é aplicável. Um caso em que não é aplicável é o do gerador zero -> nat que não contém quaisquer argumentos. Repare-se ainda que o AtomicSort é agora um objecto da classe ExistsMatchClass do DSLTrans.

```
1 nat(zero).
2 nat(suc(X)):-nat(X).
```

Listagem 4.26: Regra: Corpo da Cláusula - Textual

Após a aplicação desta regra tem-se finalmente a transformação de ambos os geradores num modelo MPrologTR, que após ser transformado num modelo MProlog (sem

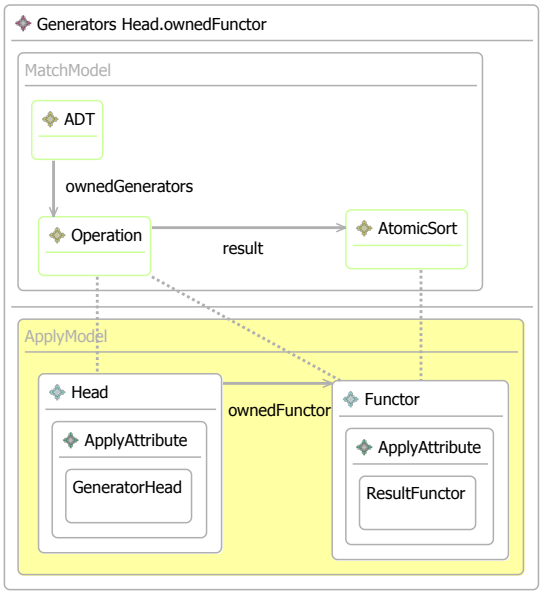

Figura 4.19: Regra: OwnedFunctor da cabeça - Visual

```
1  rule 'GenClauseHead'
2      match with
3          m264: any SATEL.APN.adtmm::
               Operation
4          m265: any SATEL.APN.adtmm::ADT
5          subject to
6              m265 --(ownedGenerators)->
                   m264
7      apply
8          m266: mprologTermReference::
               Clause(
9                      self = '
                           GeneratorClause'
10                 )
11         m267: mprologTermReference::Head
               (
12                     self = '
                           GeneratorHead'
13                 )
14         subject to
15             m266 --(ownedHead)-> m267
16     restrictions
17         m266 derived from m264
18         m267 derived from m264
19 end rule
```

Listagem 4.23: Regra: OwnedFunctor da cabeça - Textual

referências), de serem resolvidos os nomes das variáveis (##var##), e ser transformado em texto, resulta no contéudo da listagem 4.26.

Dá-se assim por concluída esta transformação dos geradores do ADT dos naturais.

4.8 Conclusões

Neste capítulo apresentou-se detalhadamente a abordagem proposta. Realçou-se que foram previamente abordadas outras soluções que foram descartadas por poderem comprometer o trabalho que se pretendia para esta dissertação. Apresentou-se uma solução metamodelada da linguagem Prolog com o fim de satisfazer a natureza de orientação a modelos em que este trabalho acenta. Mostrou-se como obter um modelo Prolog a partir do modelo SATEL⊕EAPN através de regras de transformação especificadas em DSLTrans e foi ainda abordada a técnica de obtenção dos oráculos a partir dos testes abstractos gerados.

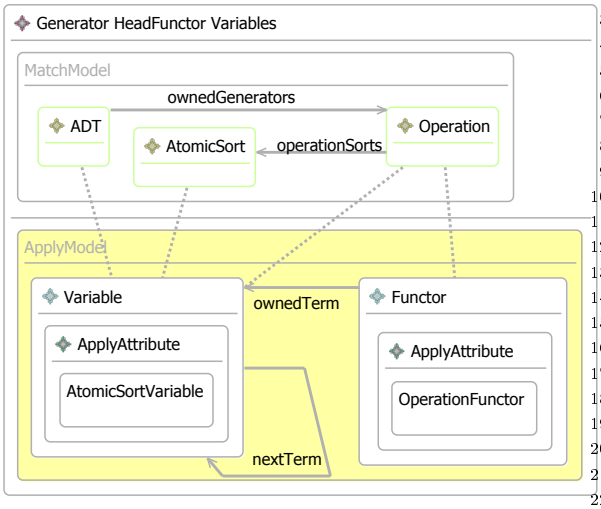

Figura 4.20: Regra: Variáveis do functor da cabeça - Visual

```
rule 'Generator HeadFunctor Variables'
 match with
  m273: any SATEL.APN.adtmm::Operation
    m274: any SATEL.APN.adtmm::AtomicSort
    m275: any SATEL.APN.adtmm::ADT
    subject to
     m273 --(operationSorts)-> m274
     m275 --(ownedGenerators)-> m273
    apply
     m276: mprologTermReference::Variable(
                self = 'AtomicSortVariable'
                   )
     m277: mprologTermReference::Functor(
                self= 'OperationFunctor'
                   )
     subject to
                m277 --(ownedTerm)-> m276
                m276 --(nextTerm)-> m276

    restrictions
     m276 derived from m274
     m277 derived from m273
     m276 derived from m275
     m276 derived from m273
end rule
```

Listagem 4.24: Regra: Variáveis do functor da cabeça - Textual

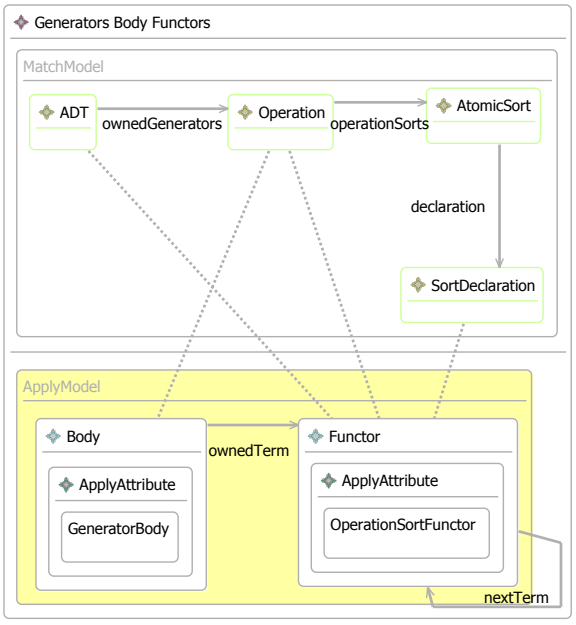

Figura 4.21: Regra: Functores do corpo da cláusula - Visual

```
rule 'Generators Body Functors'
 match with
  m278: any SATEL.APN.adtmm::Operation
    m279: any SATEL.APN.adtmm::AtomicSort
    m280: any SATEL.APN.adtmm::
         SortDeclaration
    m281: any SATEL.APN.adtmm::ADT
    subject to
     m278 --(operationSorts)-> m279
     m279 --(declaration)-> m280
     m281 --(ownedGenerators)-> m278
    apply
     m282: mprologTermReference::Body(
                self = 'GeneratorBody'
                )
     m283: mprologTermReference::Functor(
                self= 'OperationSortFunctor'
                )
    subject to
        m282 --(ownedTerm)-> m283
        m283 --(nextTerm)-> m283
    restrictions
     m282 derived from m278
     m283 derived from m281
     m283 derived from m280
     m283 derived from m278
    end rule
```

Listagem 4.25: Regra: Functores do corpo da cláusula - Textual

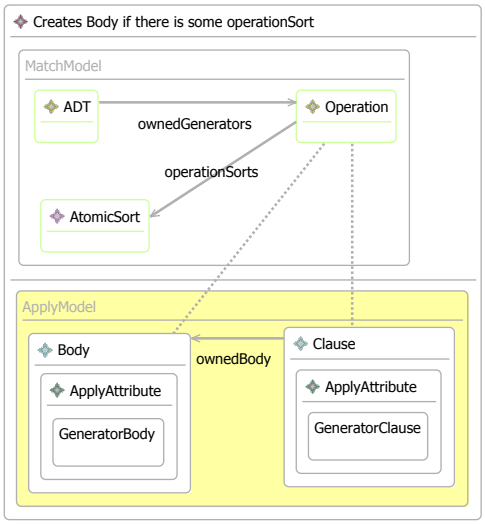

Figura 4.22: Regra: Corpo da Cláusula - Visual

```
1  rule 'Creates Body if there is some
       operationSort'
2   match with
3    m284: any SATEL.APN.adtmm::Operation
4      m285: existing SATEL.APN.adtmm::
           AtomicSort
5      m286: any SATEL.APN.adtmm::ADT
6
7      subject to
8       m284 --(operationSorts)-> m285
9       m286 --(ownedGenerators)-> m284
10
11     apply
12       m287: mprologTermReference::Clause(
13                     self = 'GeneratorClause'
14                     )
15       m288: mprologTermReference::Body(
16                         self = '
                               GeneratorBody'
17                     )
18         subject to
19                 m287 --(ownedBody)-> m288
20     restrictions
21             m288 derived from m284
22             m287 derived from m284
23     end rule
```

Listagem 4.27: Regra em sintaxe textual

5

Exemplo ilustrativo

Este capítulo serve para demonstrar a aplicação da técnica apresentada no capítulo anterior.

Para fins ilustrativos presuma-se a existência de um componente de *software* utilizado numa fábrica industrial que tem como propósito fazer a contagem de troncos de árvore que entram na máquina de processamento de madeira em toros. Sabe-se que a máquina não suporta mais de 10 troncos simultâneamente, pelo que ao décimo tronco tem de ser emitido um alarme para que a passadeira rolante pare. Por outro lado sabe-se que a madeira não vem sempre ao mesmo ritmo, apresentando latência elevada e, nesse sentido, o funcionário de controlo junto à entrada da máquina tem que cuidar para que nem a máquina, nem a passadeira passem tempos elevados sem produzir pelo que, em casos de grande latência, ele deve apertar o botão para que a máquina comece a trabalhar. Nesse momento a máquina inicia o funcionamento com a madeira que tem, a passadeira pára e é afixado e registado o numero de toros que estão em processamento. Obviamente quando o alarme dispara são também registados e acumulados à contabilidade os 10 troncos num outro componente que interpreta o *output* deste.

Antes da entrega do componente quer-se testar se este se comporta conforme os requisitos e para isso utilizar-se-á a abordagem apresentada na tese. É importante que o componente conte correctamente nas fases de intervenção humana, pelo que se quer aferir se o contador, quando *resetado*, lança para o exterior o número exacto de toros.

Obviamente este componente pode ser modelado através do modelo de contador da figura 5.1, que já tem vindo a ser utilizado. A partir deste modelo gerou-se testes

de acordo com as intenções de teste modeladas na listagem 5.5.

Nas listagens 5.1, 5.2 e 5.3 são definidas três álgebras (respectivamente naturais, booleanos, e listas que embora não sejam usadas, servem aqui para exemplificação da transformação de axiomas com condições). Encontra-se ainda na listagem 5.4 a especificação textual da EAPN apresentada na figura 5.1.

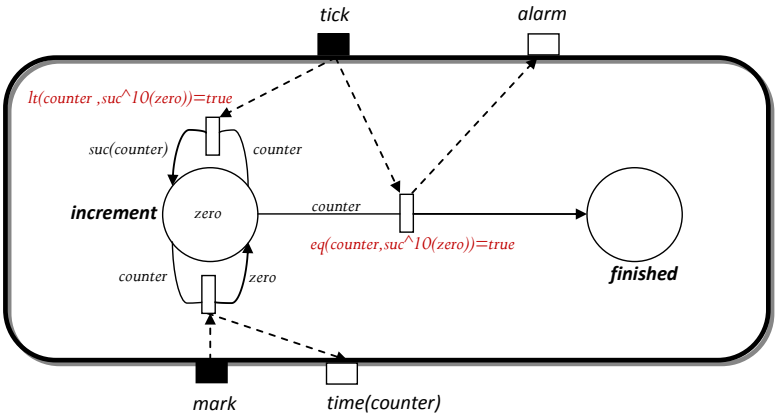

Figura 5.1: Rede de Petri Algébrica de um Contador

Na listagem 5.1 encontra-se a especificação dos naturais com dois geradores (já introduzidos anteriormente) e um conjunto de operações básicas sobre naturais algébricos e respectivos axiomas.

Listagem 5.1: Especificação de modelo: Álgebra dos naturais

```
1  TestIntentionSet mytestintention
2      Algebras
3
4          ADT naturals
5          Sort nat
6          Generators
7              zero -> nat ;
8              suc : nat -> nat ;
9          Operations
10             plus : nat, nat -> nat ;
11             sum : nat, nat -> nat ;
12             eq : nat, nat -> bool;
13             lt : nat, nat -> bool ;
14         Axioms
15             sum($X, zero ) = zero ;
16             sum($X, suc($Y))= suc(sum($X,$Y)) ;
17             eq(zero, suc($X)) = false;
18             eq(suc($Y), zero) = false;
19             eq(zero,zero) = true;
20             eq(suc($X), suc($Y)) = eq($X,$Y);
21             lt (zero, suc($X)) = true;
22             lt (suc($X), zero) = false;
23             lt (suc($X), suc($Y)) = lt($X,$Y);
24         Where
```

```
25
26              X : nat ;
27              Y : nat     ;
28
29      ...
```

A listagem 5.2 define uma álgebra simples para os booleans. Apenas são definidos dois geradores (valores de verdade e falsidade) e uma operação de negação.

Listagem 5.2: Especificação de modelo: Álgebra dos Booleans

```
1  ...
2  ADT Boolean
3      Sort bool
4      Generators
5          false -> bool;
6          true -> bool;
7      Operations
8          neg : bool -> bool;
9       Axioms
10         neg(false) = true ;
11         neg(true) = false;
12 ...
```

Na listagem 5.3 é definida uma álgebra para listas. Como referido este ADT não é utilizado directamente na geração de testes para este modelo, porém, como utiliza construções mais sofisticadas na especificação dos axiomas pareceu revelar-se importante para mostrar a potencialidade da ferramenta e da transformação. Nesta álgebra existem dois geradores, um para a lista vazia (linha 5), e outro que coloca um elemento à cabeça de uma lista existente (linha 6). Definiram-se duas operações e respectivos axiomas para calcular, respectivamente, o tamanho da lista (linha 9) e o número de ocorrências de um elemento na mesma (linha 10).

Listagem 5.3: Especificação de modelo: Álgebra das Lists

```
1      ...
2  ADT List
3  Sort list
4  Generators
5      nil -> list ;
6      cons : nat ,list -> list;
7
8  Operations
9      size : list -> nat ;
10     nbOcurr : nat ,list -> nat;
11
12 Axioms
13     eq($E,$H)=true => nbOcurr($E, cons($H,$L)) = suc(nbOcurr($E, $L)) ;
14     eq($E,$H)=false  => nbOcurr($E, cons($H,$L)) = nbOcurr($E, $L) ;
15     size (nil) = zero ;
16     size(cons($H,$L)) = suc(size($L)) ;
17     nbOcurr( $E , nil)=zero;
18 Where
```

```
19      H: nat ;
20      L : list ;
21      E: nat ;
22  End Algebras ;
23      ...
```

Na listagem 5.4 foi definido o foco das intenções de teste - a EAPN apresentada na figura. Primeiramente são definidos os *places* Inc e Finished (linha 5 e 6), onde são definidos *tokens* para Inc: um MultiSet com o *token* algébrico natural zero. É também possível especificar o sort do *place* neste ponto. Em seguida definiram-se os arcos entre os *places* e as transições anotados com os respectivos pesos através de *MultiSets*. Especificaram-se as transições indicando um nome, os métodos que as despoletam e as *gates* de observação que lhe estão afectas. No caso das transições 'increment' e 'alarmTrigger' foram também especificadas as respectivas guardas concordando com o disparo se a variável Counter excede ou iguala $suc^{10}(Counter)$.

Listagem 5.4: Especificação de modelo: APN

```
1       ...
2   Focus
3       APN Counter
4           Places
5            Place Inc tokens [ zero ]
6            Place Finished
7           Arcs
8            Arc Inc2increment  { Inc --> increment tokens [ $Counter ]  }
9            Arc increment2Inc  {  increment --> Inc tokens  [ suc($Counter)] }
10
11           Arc reset2Inc { reset --> Inc tokens [$Counter]}
12           Arc Inc2reset { Inc --> reset tokens [zero]}
13
14           Arc Inc2alarmTrigger { Inc --> alarmTrigger tokens [$Counter ]}
15           Arc alarmTrigger2Inc { alarmTrigger --> Finished }
16
17          Transitions
18           Transition increment {
19              Guard { lt($Counter, suc^10(zero))=true }
20              MethodCalls [tick]
21            }
22
23           Transition reset {
24              GateCalls [time($Counter) ]
25              MethodCalls [mark]
26            }
27
28            Transition alarmTrigger {
29              Guard { eq($Counter, suc^10(zero))=true }
30              GateCalls [alarm]
31              MethodCalls [tick]
32            }
33
34          Methods [tick, mark ]
35          Gates   [alarm, time]
```

```
36
37          Where
38              Counter : nat;
39
40  End Focus ;
41    ...
```

Por fim, são definidas as intenções de teste que têm vindo a ser utilizadas na listagem 5.5. Atente-se que a variável Counter é, neste exemplo, restringida entre $zero$ e $suc^3(zero)$, por uma questão de simplificar a apresentação de resultados.

Listagem 5.5: Especificação de modelo: Intenções de teste

```
1   ...
2   Interface
3           TestIntention TickIntention;
4           TestIntention MarkIntention;
5   Body
6       Variables
7        t : primitiveHML    ;
8        Counter : nat ;
9        x : primitiveInteger;
10       y : primitiveStimulation;
11      Axioms
12       HML(T) in TickIntention;
13       $t in TickIntention   , nbEvents($t)<6 => $t . HML ( { <tick> } T)  in TickIntention;
14       {lt($Counter ,suc^3(zero))}={true} , $t in TickIntention  => $t .HML({<mark with time(
             $Counter)>} T ) in MarkIntention;
15  End
```

5.1 Processo de obtenção de testes

Através da figura 5.2 revisita-se o plano para obtenção de testes. Primeiramente, após o modelo estar especificado, na etapa 1 faz-se *unfolding* das estruturas comprimidas; por exemplo, relativamente ao modelo, $suc^3(zero)$ é convertido em suc(suc(suc(zero))), os nomes das *gates* são prefixados com 'gate_' e dos métodos com 'method_' para que iniciem com letra minúscula, pois no Prolog os functores devem iniciar por letra minúscula; e ainda, caso existissem *multisets* contendo algum elemento com multiplicidade maior que 1 seriam gerados elementos iguais a esse com o fim de tornar todo o conjunto homogéneo no que diz respeito à multiplicidade.

Seguidamente na etapa 2 é feito o refinamento do modelo. No modelo, por exemplo no terceiro axioma das intenções de teste, a primeira condição $(lt(Counter, suc^3(zero)) = true)$ terá a sua referência a apontar para a seguinte condição $\mathrm{t\ in\ TickIntention}$. Isto é necessário para que o DSLTrans possa efectivamente processar listas.

Na etapa 3 seria gerado o modelo PrologTR a partir do modelo já refinado e nas etapas 4,5 e 6 obter-se-ia, por fim, o código Prolog tal como é apresentado na secção

seguinte.

Na posse do código Prolog geram-se os testes, e anotam-se com o oráculo nas etapas 8 e 9 respectivamente.

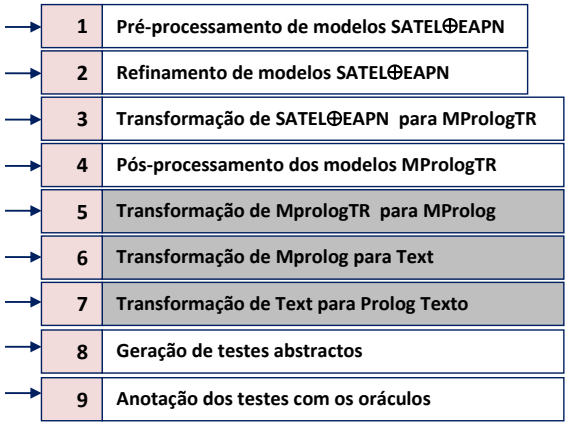

Figura 5.2: Etapas do processo para obtenção de testes

5.2 Obtenção do código Prolog

Após a especificação do modelo através do editor implementado e do pré- processamento necessário correu-se as transformações tendo em vista a obtenção do código Prolog que gerará os testes abstractos. Com o exemplo obtiveram-se as listagens 5.6, 5.7, 5.8 e 5.9 referentes às álgebras dos naturais, booleanos e listas e às intenções de teste, respectivamente. Na listagem 5.10 obteve-se o código referente às transições e marcação inicial da EAPN.

Listagem 5.6: Prolog referente ao ADT naturals

```
1  %ADT nat
2  nat(zero).
3  nat(suc(X20)) :-  nat(X20).
4  eval(zero, zero).
5  eval(suc(X24), suc(X23)) :-  eval(X24, X23).
6  eval(sum(V1, zero), X0) :- nat(V1),
7      eval(zero, X0).
8  eval(sum(V1, suc(V2)), X1) :- nat(V1),
9      eval(suc(sum(V1, V2)), X1).
10 eval(eq(zero, suc(V1)), X2) :- nat(V1),
11     eval(false, X2).
12 eval(eq(suc(V2), zero), X3) :- nat(V2),
13     eval(false, X3).
14 eval(eq(zero, zero), X4) :-
15     eval(true, X4).
16 eval(eq(suc(V1), suc(V2)), X5) :- nat(V1), nat(V2),
17     eval(eq(V1, V2), X5).
```

```
eval(lt(zero, suc(V1)), X6) :- nat(V1),
    eval(true, X6).
eval(lt(suc(V1), zero), X7) :-  nat(V1),
    eval(false, X7).
eval(lt(suc(V1), suc(V2)), X8) :- nat(V1), nat(V2),
    eval(lt(V1, V2), X8).
```

Listagem 5.7: Prolog referente ao ADT boolean

```
%ADT boolean
eval(neg(false), X9) :-
    eval(true, X9).
eval(neg(true), X10) :-
    eval(false, X10).
bool(false).
bool(true).
eval(false, false).
eval(true, true).
```

Listagem 5.8: Prolog referente ao ADT list

```
list(nil).
list(cons(X21, X22)) :-
    nat(X21), list(X22).
eval(nil, nil).
eval(cons(X27, X28), cons(X25, X26)) :-
    eval(X27, X25), eval(X28, X26).

eval(nbOcurr(V6, cons(V4, V5)), X11) :- nat(V6), nat(V4), list(V5),
    eval(eq(V6, V4), X14),
    eval(true, X14),
    eval(eq(suc(V6), zero), X13),
    eval(false, X12), (X13 \= X12),
    eval(suc(nbOcurr(V6, V5)), X11).

eval(nbOcurr(V6, cons(V4, V5)), X15) :- nat(V6), nat(V4), list(V5),
    eval(eq(V6, V4), X16), eval(false, X16), eval(nbOcurr(V6, V5), X15).
eval(size(nil), X17) :-
    eval(zero, X17).
eval(size(cons(V4, V5)), X18) :- nat(V4), list(V5),
    eval(suc(size(V5)), X18).
eval(nbOcurr(V6, nil), X19) :-  nat(V6),
    eval(zero, X19).
```

Listagem 5.9: Prolog referente às intenções de teste

```
%Intenções de Teste
in([top], 'TickIntention').
in([X,(tick,null),top],'TickIntention'):-
      eval(in(F,'TickIntention'), true),flatten(F,X), eval(boolLT(nbEvents(X),6),true).
in([X,(mark,time(Counter)),top],'MarkIntention'):-
      eval(in(F,'TickIntention'),true), flatten(F,X), eval(algEquality( lt( Counter,      suc(
          suc(suc(zero))) ), true), true).
initialMarcation([place('Inc', [zero]), place('Finished', [])]).
```

Listagem 5.10: Prolog referente à EAPN

```
initialMarcation([place('Inc', [zero]), place('Finished', [])]).
transition(tick, null, Marcation, NewMarcation):-
    takeout(Counter, place('Inc'),Marcation, NewMarcation1),
    putin(suc(Counter), place('Inc'),NewMarcation1, NewMarcation),
    eval(algEq(lessThan( Counter,suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(zero)))))))))) ),
        true), true).

transition(mark, time(Counter), Marcation, NewMarcation):-
  takeout(Counter, place('Inc'),Marcation, NewMarcation1),
    putin(zero, place('Inc'),NewMarcation1, NewMarcation).

transition(tick, alarm, Marcation, NewMarcation):-
  takeout(Counter, place('Inc'),Marcation, NewMarcation1),
    eval(algEq(lessThan( Counter,suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(zero)))))))))) ),
        true), true).
```

5.3 Geração de Testes

Após a geração das cláusulas Prolog puderam então ser gerados os testes abstractos no seguinte ambiente:

- Laptop (4Gb RAM, Pentium Dual Core 2.30 GHz)
- Sistema operativo Windows 7
- SWI-Prolog 5.10.4

Para obter os testes é necessário importar a implementação da semântica operacional do SATEL e fazer a seguinte interrogação, em que se limita a profundidade de pesquisa (níveis de functores aninhados) a 15.

```
call_with_depth_limit((in(X,'MarkIntention'), flatten(X,Y), reduceSolution(Y,Sol)),15,_).
```

O predicado flatten serve simplesmente para que não existam listas aninhadas e o predicado reduceSolution (listagem 5.11) serve para eliminar eventuais fórmulas top (HMLTop) que tenham ficado entre os eventos, devido à computação.

Listagem 5.11: Soluções obtidas

```
reduceSolution([],[]).
reduceSolution([top],[top]):-!.
reduceSolution([top,X|Tail],XX):-reduceSolution([X|Tail],XX),!.
reduceSolution([H|T],[H|TT]):-reduceSolution(T,TT).
```

As soluções obtidas encontram-se na listagem 5.12 e como se pode observar estão de acordo com os axiomas das intenções de teste; note-se que nenhuma solução excede os 6 eventos e a variável Counter foi instanciada com valores que não excedem $suc^3(zero)$.

Listagem 5.12: Soluções obtidas

```
1  Sol = [ (mark, time(zero)), top] ;
2  Sol = [ (mark, time(suc(zero))), top] ;
3  Sol = [ (mark, time(suc(suc(zero)))), top] ;
4  Sol = [ (tick, null), (mark, time(zero)), top] ;
5  Sol = [ (tick, null), (mark, time(suc(zero))), top] ;
6  Sol = [ (tick, null), (mark, time(suc(suc(zero)))), top] ;
7  Sol = [ (tick, null), (tick, null), (mark, time(zero)), top] ;
8  Sol = [ (tick, null), (tick, null), (mark, time(suc(zero))), top] ;
9  Sol = [ (tick, null), (tick, null), (mark, time(suc(suc(zero)))), top] ;
10 Sol = [ (tick, null), (tick, null), (tick, null), (mark, time(zero)), top] ;
11 Sol = [ (tick, null), (tick, null), (tick, null), (mark, time(suc(zero))), top] ;
12 Sol = [ (tick, null), (tick, null), (tick, null), (mark, time(suc(suc(zero)))), top] ;
13 Sol = [ (tick, null), (tick, null), (tick, null), (tick, null), (mark, time(zero)), top] ;
14 Sol = [ (tick, null), (tick, null), (tick, null), (tick, null), (mark, time(suc(zero))), top]
       ;
15 Sol = [ (tick, null), (tick, null), (tick, null), (tick, null), (mark, time(suc(suc(zero)))),
       top] ;
```

5.4 Obtenção dos oráculos

Após a obtenção dos testes calcularam-se os oráculos da forma que foi exposta no capítulo anterior, tal como apresentado na listagem 5.13.

Listagem 5.13: Testes abstractos anotados com oráculo

```
1  SolO = ([ (mark, time(zero)), top], true);
2  SolO = ([ (mark, time(suc(zero))), top], false) ;
3  SolO = ([ (mark, time(suc(suc(zero)))), top], false) ;
4  SolO = ([ (tick, null), (mark, time(zero)), top], false) ;
5  SolO = ([ (tick, null), (mark, time(suc(zero))), top], true) ;
6  SolO = ([ (tick, null), (mark, time(suc(suc(zero)))), top], false) ;
7  SolO = ([ (tick, null), (tick, null), (mark, time(zero)), top], false) ;
8  SolO = ([ (tick, null), (tick, null), (mark, time(suc(zero))), top], false) ;
9  SolO = ([ (tick, null), (tick, null), (mark, time(suc(suc(zero)))), top], true) ;
10 SolO = ([ (tick, null), (tick, null), (tick, null), (mark, time(zero)), top], false) ;
11 SolO = ([ (tick, null), (tick, null), (tick, null), (mark, time(suc(zero))), top], false) ;
12 SolO = ([ (tick, null), (tick, null), (tick, null), (mark, time(suc(suc(zero)))), top], false)
       ;
13 SolO = ([ (tick, null), (tick, null), (tick, null), (tick, null), (mark, time(zero)), top],
       false) ;
14 SolO = ([ (tick, null), (tick, null), (tick, null), (tick, null), (mark, time(suc(zero))), top
       ], false) ;
15 SolO = ([ (tick, null), (tick, null), (tick, null), (tick, null), (mark, time(suc(suc(zero))))
       , top], false) ;
```

Pode-se verificar que os oráculos foram calculados correctamente. Note-se que para as linhas 1, 5 e 9, em que oráculo é verdadeiro indicando que o teste deverá passar, o número de invocações tick ocorridas até à invocação da *gate* time corresponde ao parâmetro deste; isto é, zero invocações de tick na primeira solução, uma invocação na solução da linha número 5, e duas invocações na solução da linha número 9.

Note-se que para este exemplo foi desactivado o pré-processamento que prefixa os métodos com 'meth' e as *gates* com 'gate', com o fim de melhorar a apresentação.

5.5 Conclusões

Neste capítulo apresentou-se uma concretização da abordagem com um exemplo ilustrativo - o timer - que acompanhou a tese desde a introdução da linguagem SATEL no Capítulo 3. Serviu, deste modo, para demonstrar o processo de geração de testes abstractos anotados com oráculo a que se propunha a tese e que reflecte totalmente a abordagem Model-Based Testing.

Como discutido, a ferramenta em questão centra-se numa abordagem totalmente baseada em modelos (compreendendo técnicas de transformação de modelos, ou seja, alguns pré e pós processamentos dos modelos em causa). De facto, utilizam-se modelos para modelar o SUT e também para guiar a geração por entre os diversos traços/caminhos de execução do sistema que se queira cobrir. Tendo em conta o estado da arte, com falta de implementações concretas, considera-se ter desenvolvido uma ferramenta inovadora neste estilo de Model-Based Testing.

Numa análise mais pormenorizada, viu-se que perante este exemplo o tempo de execução da ferramenta é infinitésimal. Isto justifica-se com a dimensão do problema, que é significativamente reduzida, e pelas condições envolvidas. Obviamente com o relaxamento da profundidade de pesquisa e alterações das condições (e.g. eliminando a restrição sobre Counter) o tempo de execução é infinito e a ferramenta corrompe-se pois o Prolog não encontra um ponto de *terminus* (fixo); isto é, em certas situações o Prolog pode não conseguir concluir que a pesquisa pode ser parada (e.g. nos naturais algébricos não existe forçosamente uma relação de ordem entre os elementos; o Prolog, perante uma condição que é satisfeita para qualquer número 'inferior' a $suc^3(zero)$, não consegue concluir que existem apenas 3 alternativas, e irá continuar a procurar). Uma forma de limitar esta questão é a de apenas utilizar geradores monótonos e não ambíguos nos ADT, no entanto seria ainda necessário procurar que a semântica das operações seguisse essa regra também, e que a pesquisa tivesse isso em conta; não sendo possível, resta usar heurísticas que limitem a pesquisa.

De qualquer forma, problemas como estes são bem conhecidos da geração de testes, dado que a geração exaustiva é, em problemas relaxados e em geral, impraticável.

Trata-se de trabalho futuro tentar lidar com estes problemas de uma forma eficiente, utilizando heurísticas apropriadas e, possivelmente, estendendo o metamodelo do SATEL com construções que possam informar a ferramenta de novas condições de paragem e/ou dados que promovam a truncatura da árvore de pesquisa em nós a

menor profundidade. Isto considerando sempre uma abordagem baseada modelos.

6

Conclusões

Ao longo desta dissertação foi apresentada a semântica operacional e implementação da linguagem SATEL validando a hipótese proposta de o conseguir em conformidade com o trabalho feito previamente, de especificação formal da sua semântica de um modo denotacional. Deste modo, foi conseguida a geração de testes abstractos a partir de modelos APN com extensões mínimas; testes esses que constituem traços de execução do Sistema sob Teste.

Conseguiu-se uma implementação para a linguagem SATEL, que em detrimento de outras abordagens tentadas, e devidamente discutidas no capítulo 4 e por ser usado o Prolog como veículo de especificação dos testes e obtenção de soluções, se aproxima da especificação da semântica denotacional original. Acrescenta-se ainda que a utilização de uma sintaxe textual através do EMFText que compreende apenas algumas extensões mínimas à linguagem original, contribui também para a forte ligação à sua definição formal. Esta implementação permite-nos validar a SATEL enquanto linguagem que gera efectivamente testes abstractos. Em sumário, a implementação permite-nos fechar o ciclo da abordagem MBT, dando-nos garantias relativamente à SATEL enquanto abordagem baseada em modelos para geração de testes sendo esta, em nossa opinião, uma grande contribuição desta dissertação.

Para a obtenção da implementação foi proposta uma abordagem de desenvolvimento orientada a modelos em que os modelos SATEL$\oplus$EAPN sofrem sucessivas transformações automáticas até se obter um conjunto de cláusulas Prolog que permitem, finalmente, a pesquisa de soluções. Esta abordagem está submetida obviamente às idiosincrasias do Prolog, como a pesquisa em profundidade para a qual a nossa solução

propõe uma heurística simples de limitação.

6.1 Trabalho futuro

Ao longo da dissertação foram feitas algumas referências em relação a trabalho futuro. Não obstante de se ter implementado uma semântica operacional do SATEL, houve alguns aspectos da linguagem que não pertenceram ao foco da implementação. Assim sendo, como trabalho futuro aponta-se:

- a implementação das hipóteses de uniformidade e subuniformidade sobre variáveis;
- melhoria da obtenção de valores para as variáveis algébricas, possivelmente com algum *unfolding* prévio das estruturas algébricas, contracção dos geradores base dos ADT (i.e., limitar à partida com base nas condições dos axiomas, o primeiro valor a ser gerado), e ainda com extensão do satel, para que o engenheiro de teste possa anotar os ADT com indicações de restrição mais efectivas;
- prever as fórmulas HMLNot, e HMLAnd nos caminhos de execução;
- possível extensão da abordagem a modelos especificados em CO-OPN, à semelhança do que foi proposto originalmente;
- concretização dos testes abstractos gerados;
- tendo em vista a optimização, portar a implementação para linguagens de outros paradigmas(e.g. imperativo), com implementação implícita de algoritmos de *backtracking*, pesquisa e resolução de restrições que consigam melhorar o desempenho e que façam face às idiosincrasias do Prolog (eg. pesquisa em *depth-first search*).
- integração com o model-checker ALPiNa [1] desenvolvido pelo grupo SMV [2] da Universidade de Genebra [3]

[1] http://alpina.unige.ch
[2] https://smv.unige.ch
[3] https://www.unige.ch

Appendix

A.1 Metamodelo do SATEL em OCLinEcore [1]

```
import ecore_0 : 'http://www.eclipse.org/emf/2002/Ecore#/';

package SATEL : SATEL = 'SATEL'
{
    class Model
    {
        property testIntentionModule : TestIntentionModule[+] { !ordered,composes };
    }
    class TestIntentionModule
    {
        property testIntentionBody : TestIntentionBody[1] { !ordered,composes };
        property testIntentionInterface : TestIntentionInterface[?] { !ordered,composes };
        attribute name : String[1] { !ordered,!unique };
        property focus : apnmm::APN[1] { composes };
        property algebra : adtmm::ADT[*] { composes };
    }
    class TestIntentionInterface
    {
        property intention : IntentionDec[+] { !ordered,composes };
    }
    class IntentionDec
    {
        attribute name : String[1] { !ordered,!unique };
    }
    class TestIntentionBody
    {
        property axiomDeclaration : AxiomDeclaration[1] { !ordered,composes };
        property variableDeclaration : VariableDeclarations::VariableDeclaration[?] { !ordered
            ,composes };
```

[1] http://wiki.eclipse.org/MDT/OCLinEcore

```
29      }
30      class AxiomDeclaration
31      {
32          property axiom : Axiom[+] { !ordered,composes };
33      }
34      class Axiom
35      {
36          property condition : Condition[?] { !ordered,composes };
37          property inclusion : Inclusion[1] { !ordered,composes };
38      }
39      class Inclusion extends ConditionAtom
40      {
41          property hmlTerm : HMLFormula::HMLTerm[1] { !ordered,composes };
42          property _'in' : IntentionDec[1] { !ordered,!unique };
43      }
44      class Condition
45      {
46          property domainQuantifier : DomainQuantifier[?] { !ordered,composes };
47          property conditionBody : ConditionBody[1] { !ordered,composes };
48      }
49      class DomainQuantifier { abstract }
50      {
51          property list : NameList[1] { !ordered,composes };
52      }
53      class UniformityList extends DomainQuantifier;
54      class SubuniformityList extends DomainQuantifier;
55      class ConditionBody
56      {
57          property conditionAtom : ConditionAtom[+] { composes };
58      }
59      class ConditionAtom { abstract }
60      {
61          property next : ConditionAtom[?];
62      }
63      class NameList
64      {
65          property nameRef : VariableDeclarations::VariableDec[+] { !ordered,!unique };
66      }
67      package HMLFormula : HMLFormula = 'SATEL.HMLFormula'
68      {
69          class HMLTerm
70          {
71              property hmlFormula : HMLFormula[+] { !ordered,composes };
72          }
73          class HMLFormula { abstract }
74          {
75              property next : HMLFormula[?];
76          }
77          class HMLFormulaHMLFormulaContent extends HMLFormula
78          {
79              property hmlFormulaContent : HMLFormulaContent[1] { composes };
80          }
81          class HMLFormulaPrimitiveHMLVarDec extends HMLFormula
82          {
83              property primitiveHMLVarDec : VariableDeclarations::PrimitiveHMLVarDec[1];
84          }
85          class HMLFormulaContent { abstract };
```

```
86          class HMLNext extends HMLFormulaContent
87          {
88              property hmlFormulaContent : HMLFormulaContent[1] { !ordered,composes };
89              property hmlEvent : HMLEvent[1] { !ordered,composes };
90          }
91          class HMLNot extends HMLFormulaContent
92          {
93              property hmlFormulaContent : HMLFormulaContent[1] { !ordered,composes };
94          }
95          class HMLAnd extends HMLFormulaContent
96          {
97              property hmlFormulaContentL : HMLFormulaContent[1] { !ordered,composes };
98              property hmlFormulaContentR : HMLFormulaContent[1] { !ordered,composes };
99          }
100         class HMLTop extends HMLFormulaContent;
101         class HMLEvent
102         {
103             property inputTerm : SynchronizationInputTerm[1] { !ordered,composes };
104             property outputTerm : SynchronizationOutputTerm[?] { !ordered,composes };
105         }
106         class SynchronizationTerm { abstract };
107         class SynchronizationInputTerm extends SynchronizationTerm { abstract };
108         class SynchronizationEventInputTerm extends SynchronizationInputTerm
109         {
110             property event : InputEvent[1];
111             property parameters : Parameter[?] { composes };
112         }
113         class SynchronizationOutputTerm extends SynchronizationTerm { abstract };
114         class SynchronizationEventOutputTerm extends SynchronizationOutputTerm
115         {
116             property event : OutputEvent[1];
117             property parameters : Parameter[?] { composes };
118         }
119         class WPrimitiveObservationVarDec extends SynchronizationOutputTerm
120         {
121             property primitiveObservation : VariableDeclarations::PrimitiveObservationVarDec
                    [1];
122         }
123         class WPrimitiveStimulationVarDec extends SynchronizationInputTerm
124         {
125             property primitiveStimulation : VariableDeclarations::PrimitiveStimulationVarDec
                    [1];
126         }
127         class Parameter
128         {
129             property value : algterms::AlgebraicTerm[1] { composes };
130             property next : Parameter[?] { composes };
131         }
132         class InputEvent { abstract };
133         class OutputEvent { abstract };
134     }
135     package AlgebraicExpressions : AlgebraicExpressions = 'SATEL.AlgebraicExpressions'
136     {
137         class AlgebraicEquality extends SATEL::ConditionAtom { abstract };
138         class AlgEquality extends AlgebraicEquality
139         {
140             property algebraicTermL : algterms::AlgebraicTerm[1] { !ordered,composes };
```

```
141             property algebraicTermR : algterms::AlgebraicTerm[1] { !ordered,composes };
142         }
143         class SyncEquality extends AlgebraicEquality
144         {
145             property synchronizationTermL : HMLFormula::SynchronizationTerm[1] { !ordered,
                    composes };
146             property synchronizationTermR : HMLFormula::SynchronizationTerm[1] { !ordered,
                    composes };
147         }
148         class HMLEquality extends AlgebraicEquality
149         {
150             property hmlTermL : HMLFormula::HMLTerm[1] { !ordered,composes };
151             property hmlTermR : HMLFormula::HMLTerm[1] { !ordered,composes };
152         }
153         class BooleanEquality extends AlgebraicEquality
154         {
155             property booleanTermR : booleanterms::BooleanTerm[1] { !ordered,composes };
156             property booleanTermL : booleanterms::BooleanTerm[1] { !ordered,composes };
157         }
158         class ArithmeticEquality extends AlgebraicEquality
159         {
160             property arithmeticTermL : arithmeticterms::ArithmeticTerm[1] { !ordered,composes
                    };
161             property arithmeticTermR : arithmeticterms::ArithmeticTerm[1] { !ordered,composes
                    };
162         }
163         package booleanterms : booleanterms = 'SATEL.AlgebraicExpressions.booleanterms'
164         {
165             class BooleanTerm extends SATEL::ConditionAtom { abstract };
166             class Not extends BooleanTerm
167             {
168                 property booleanTerm : BooleanTerm[1] { !ordered,composes };
169             }
170             class _'Sequence' extends BooleanTerm
171             {
172                 property hmlTerm : HMLFormula::HMLTerm[1] { !ordered,composes };
173             }
174             class Positive extends BooleanTerm
175             {
176                 property hmlTerm : HMLFormula::HMLTerm[1] { !ordered,composes };
177             }
178             class Trace extends BooleanTerm
179             {
180                 property hmlTerm : HMLFormula::HMLTerm[1] { !ordered,composes };
181             }
182             class BooleanVariable extends BooleanTerm
183             {
184                 property booleanVariable : VariableDeclarations::PrimitiveBooleanVarDec[1] { !
                        ordered,!unique };
185             }
186             class BooleanValue extends BooleanTerm
187             {
188                 attribute value : Boolean[1] { !ordered,!unique };
189             }
190             class BOPAnd extends BooleanTerm
191             {
192                 property booleanTermL : BooleanTerm[1] { !ordered,composes };
```

```
193                 property booleanTermR : BooleanTerm[1] { !ordered,composes };
194             }
195             class BOPOr extends BooleanTerm
196             {
197                 property booleanTermL : BooleanTerm[1] { !ordered,composes };
198                 property booleanTermR : BooleanTerm[1] { !ordered,composes };
199             }
200             class Equals extends BooleanTerm
201             {
202                 property arithmeticTermR : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
203                 property arithmeticTermL : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
204             }
205             class NotEquals extends BooleanTerm
206             {
207                 property arithmeticTermR : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
208                 property arithmeticTermL : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
209             }
210             class GT extends BooleanTerm
211             {
212                 property arithmeticTermR : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
213                 property arithmeticTermL : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
214             }
215             class LT extends BooleanTerm
216             {
217                 property arithmeticTermR : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
218                 property arithmeticTermL : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
219             }
220             class GTE extends BooleanTerm
221             {
222                 property arithmeticTermR : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
223                 property arithmeticTermL : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
224             }
225             class LTE extends BooleanTerm
226             {
227                 property arithmeticTermR : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
228                 property arithmeticTermL : arithmeticterms::ArithmeticTerm[1] { !ordered,
                        composes };
229             }
230         }
231         package arithmeticterms : arithmeticterms = 'SATEL.AlgebraicExpressions.
                arithmeticterms'
232         {
233             class ArithmeticTerm { abstract };
234             class IntegerVariable extends ArithmeticTerm
235             {
```

```
236                 property integerVariable : VariableDeclarations::PrimitiveIntegerVarDec[1] { !
                        ordered,!unique };
237             }
238             class IntegerValue extends ArithmeticTerm
239             {
240                 attribute value : ecore_0::EInt[1] { !ordered,!unique };
241             }
242             class BOPPlus extends ArithmeticTerm
243             {
244                 property arithmeticTermL : ArithmeticTerm[1] { !ordered,composes };
245                 property arithmeticTermR : ArithmeticTerm[1] { !ordered,composes };
246             }
247             class BOPMinus extends ArithmeticTerm
248             {
249                 property arithmeticTermL : ArithmeticTerm[1] { !ordered,composes };
250                 property arithmeticTermR : ArithmeticTerm[1] { !ordered,composes };
251             }
252             class BOPTimes extends ArithmeticTerm
253             {
254                 property arithmeticTermR : ArithmeticTerm[1] { !ordered,composes };
255                 property arithmeticTermL : ArithmeticTerm[1] { !ordered,composes };
256             }
257             class BOPDiv extends ArithmeticTerm
258             {
259                 property arithmeticTermL : ArithmeticTerm[1] { !ordered,composes };
260                 property arithmeticTermR : ArithmeticTerm[1] { !ordered,composes };
261             }
262             class NBEvents extends ArithmeticTerm
263             {
264                 property hmlTerm : HMLFormula::HMLTerm[1] { !ordered,composes };
265             }
266             class Depth extends ArithmeticTerm
267             {
268                 property hmlTerm : HMLFormula::HMLTerm[1] { !ordered,composes };
269             }
270             class UOPMinus extends ArithmeticTerm
271             {
272                 property arithmeticTerm : ArithmeticTerm[1] { !ordered,composes };
273             }
274         }
275         package algterms : algterms = 'SATEL.AlgebraicExpressions.algterms'
276         {
277             class AlgebraicTerm { abstract }
278             {
279                 property next : AlgebraicTerm[?];
280             }
281             class VariableRef extends AlgebraicTerm
282             {
283                 property var : VariableDeclarations::AlgebraicSortVarDec[1] { !ordered,!unique
                        };
284             }
285             class AbstractCompositeTerm extends AlgebraicTerm { abstract }
286             {
287                 property terms : AlgebraicTerm[*] { !ordered,composes };
288                 property op : adtmm::Operation[1] { !ordered,!unique };
289                 attribute iter : ecore_0::EInt[?] = '0';
290             }
```

```
291             class CompositeTerm extends AbstractCompositeTerm;
292         }
293     }
294     package VariableDeclarations : VariableDeclarations = 'SATEL.VariableDeclarations'
295     {
296         class VariableDeclaration
297         {
298             property variable : VariableDec[*] { !ordered,composes };
299         }
300         class VariableDec { abstract }
301         {
302             attribute name : String[1] { !ordered,!unique };
303         }
304         class PrimitiveHMLVarDec extends VariableDec
305         {
306             property type : Types::PrimitiveHML[?] { !ordered,composes };
307         }
308         class PrimitiveStimulationVarDec extends VariableDec
309         {
310             property type : Types::PrimitiveStimulation[?] { !ordered,composes };
311         }
312         class PrimitiveObservationVarDec extends VariableDec
313         {
314             property type : Types::PrimitiveObservation[?] { !ordered,composes };
315         }
316         class PrimitiveIntegerVarDec extends VariableDec
317         {
318             property type : Types::PrimitiveInteger[?] { !ordered,composes };
319         }
320         class PrimitiveBooleanVarDec extends VariableDec
321         {
322             property type : Types::PrimitiveBoolean[?] { !ordered,composes };
323         }
324         class AlgebraicSortVarDec extends VariableDec
325         {
326             property type : adtmm::SortDeclaration[1] { !ordered };
327         }
328         package Types : Types = 'SATEL.VariableDeclarations.Types'
329         {
330             class VarDecType { abstract };
331             class PrimitiveHML extends VarDecType;
332             class PrimitiveStimulation extends VarDecType;
333             class PrimitiveObservation extends VarDecType;
334             class PrimitiveInteger extends VarDecType;
335             class PrimitiveBoolean extends VarDecType;
336         }
337     }
338     package APN : APN = 'SATEL.APN'
339     {
340         package apnmm : apnmm = 'SATEL.APN.apnmm'
341         {
342             class APN extends environmentmm::Environment
343             {
344                 property ownedPlaces : Place[+] { composes };
345                 property ownedArcs : Arc[*] { composes };
346                 property ownedVariables : adtmm::Variable[*] { composes };
347                 attribute name : String[1];
```

```
348                 property methods : Method[*] { composes };
349                 property gates : Gate[*] { composes };
350                 property ownedTransitions : Transition[+] { composes };
351             }
352             class Node { abstract }
353             {
354                 attribute name : String[1];
355             }
356             class Arc
357             {
358                 property from : Node[1];
359                 property to : Node[1];
360                 property ownedArcMultiset : multisetmm::Multiset[1] { composes };
361                 attribute Name : String[?];
362             }
363             class Place extends Node
364             {
365                 property ownedPlaceMultiset : multisetmm::Multiset[1] { composes };
366                 property sort : adtmm::Sort[1] { composes };
367             }
368             class Transition extends Node
369             {
370                 property ownedGuard : guardmm::Guard[?] { composes };
371                 property gateCalls : GateCall[*] { composes };
372                 property methodCall : MethodCall[?] { composes };
373             }
374             class Method extends HMLFormula::InputEvent
375             {
376                 attribute name : String[1];
377             }
378             class Gate extends HMLFormula::OutputEvent
379             {
380                 attribute name : String[1];
381             }
382             class GateCall
383             {
384                 property gate : Gate[1];
385                 property params : adtmm::Term[*] { composes };
386                 property next : GateCall[?];
387             }
388             class MethodCall
389             {
390                 property method : Method[1];
391                 property params : adtmm::Term[*] { composes };
392             }
393             annotation Static_checks('0' = 'an arc must go from a transition to a place, or
                    from a place to a transition', '1' =
394                 'Check that there is at least one place', '2' = 'Check arc sort', '3' =
395                 'Check that the variables in the post conditions and guards are present in the
                        preconditions');
396         }
397         package environmentmm : environmentmm = 'SATEL.APN.environmentmm'
398         {
399             class Environment { abstract };
400         }
401         package guardmm : guardmm = 'SATEL.APN.guardmm'
402         {
```

```
403             class Guard extends environmentmm::Environment
404             {
405                 property ownedEquations : adtmm::AbstractEquation[+] { composes };
406                 property ownedVariables : adtmm::Variable[*] { composes };
407             }
408         }
409         package adtmm : adtmm = 'SATEL.APN.adtmm'
410         {
411             class ADT extends environmentmm::Environment
412             {
413                 property ownedSorts : SortDeclaration[*] { composes };
414                 property ownedOperations : Operation[*] { composes };
415                 property ownedGenerators : Operation[*] { composes };
416                 property ownedVariables : Variable[*] { composes };
417                 property ownedAxioms : CondEquation[*] { composes };
418                 attribute name : String[1];
419             }
420             class Sort extends AbstractSort;
421             class Operation extends AbstractOperation
422             {
423                 property operationSorts : Sort[*] { composes };
424                 property result : Sort[1] { composes };
425             }
426             class Variable
427             {
428                 attribute name : String[1];
429                 property variableSort : Sort[?] { composes };
430             }
431             class CondEquation
432             {
433                 property ownedConditions : AbstractEquation[*] { composes };
434                 property ownedEquation : Equation[1] { composes };
435             }
436             class Term { abstract }
437             {
438                 property next : Term[?];
439             }
440             class VariableRef extends Term
441             {
442                 property variable : Variable[1];
443             }
444             class CTerm extends Term
445             {
446                 attribute iter : ecore_0::EInt[?];
447                 property ownedTerms : Term[*] { composes };
448                 property op : Operation[1];
449             }
450             class AbstractEquation { abstract }
451             {
452                 property ownedRightTerm : Term[1] { composes };
453                 property ownedLeftTerm : Term[1] { composes };
454                 property next : AbstractEquation[?];
455             }
456             class Equation extends AbstractEquation;
457             class Inequation extends AbstractEquation;
458             class AbstractSort { abstract,interface };
459             class AtomicSort extends Sort
```

```
460             {
461                 property declaration : SortDeclaration[1];
462             }
463             class SortDeclaration
464             {
465                 attribute name : String[1];
466             }
467             class AbstractOperation { abstract,interface }
468             {
469                 attribute name : String[1];
470             }
471             class AbstractGenericOp extends AbstractOperation { abstract,interface };
472             annotation StaticChecks('1' = 'Check that a composite term has as many subterms as
                    its Operation has sorts.', '2' =
473                 'Check that the iter theory is only applied if there is only one subterm.', '3
                    ' =
474                 'Check that thare are not any free variables in the CondEquation conditions.',
                    '4' =
475                 'Check that there are only renames if there is an instantiation, and which are
                    the correct renames.');
476             annotation Notes_about_visitors('' =
477                 'AbstractSort, AbstractGenericSort, AbstractOperation, AbstracGenericOperation
                    and Instantiable do not implement any visitor');
478         }
479         package multisetmm : multisetmm = 'SATEL.APN.multisetmm'
480         {
481             class Multiset extends environmentmm::Environment
482             {
483                 property msSort : adtmm::Sort[?] { composes };
484                 property ownedNumOfTerms : NumOfTerms[*] { composes };
485                 property ownedVariables : adtmm::Variable[*] { composes };
486             }
487             class NumOfTerms
488             {
489                 attribute card : ecore_0::EInt[1] = '1';
490                 property ownedElement : MSElement[1] { composes };
491                 property next : NumOfTerms[?];
492             }
493             class MSElement { abstract };
494             class TermReference extends MSElement
495             {
496                 property termReferenced : adtmm::Term[1] { composes };
497             }
498         }
499     }
500 }
```

A.2 Sintaxe concreta do SATEL em EMFText [2]

```
SYNTAXDEF SATEL
FOR <SATEL>
START Model

IMPORTS {
    SATEL.HMLFormula:<SATEL.HMLFormula>
    SATEL.AlgebraicExpressions:<SATEL.AlgebraicExpressions>
    SATEL.AlgebraicExpressions.booleanterms:<SATEL.AlgebraicExpressions.booleanterms>
    SATEL.AlgebraicExpressions.arithmeticterms:<SATEL.AlgebraicExpressions.arithmeticterms>
    SATEL.AlgebraicExpressions.algterms:<SATEL.AlgebraicExpressions.algterms>
    SATEL.VariableDeclarations:<SATEL.VariableDeclarations>
    SATEL.VariableDeclarations.Types:<SATEL.VariableDeclarations.Types>
    SATEL.APN.apnmm:<SATEL.APN.apnmm>
    SATEL.APN.environmentmm:<SATEL.APN.environmentmm>
    SATEL.APN.guardmm:<SATEL.APN.guardmm>
    SATEL.APN.adtmm:<SATEL.APN.adtmm>
    SATEL.APN.multisetmm:<SATEL.APN.multisetmm>
}


OPTIONS {
    usePredefinedTokens = "false";
    defaultTokenName = "IDENTIFIER";
    generateCodeFromGeneratorModel = "true";
}


TOKENS {

  DEFINE WHITESPACE $(' ' | '\t' | '\f')$;
  DEFINE LINEBREAK $('\r\n' | '\r' | '\n')$;
  DEFINE IDENTIFIER $('A'..'Z' | 'a'..'z' | '_')('A'..'Z' | 'a'..'z' | '0'..'9' | '-'| '_')*$;
  DEFINE INTEGER $('-')?('1'..'9')('0'..'9')*|'0'$;
  DEFINE FLOAT $('-')?(('1'..'9') ('0'..'9')* | '0') '.' ('0'..'9')+ $;
  DEFINE SL_COMMENT $'//'(~('\n'|'\r'|'\ufffd'))*$;
}

TOKENSTYLES {
  "SL_COMMENT" COLOR #5CB627;
  "INTEGER" COLOR #0000FF;
  "$" COLOR #00AAFF;
  "IDENTIFIER" COLOR #0080FF;
}

RULES {
  Model ::= (testIntentionModule)+;
  TestIntentionModule ::= "TestIntentionSet" name[IDENTIFIER] !1
              "Algebras" (!1 algebra)* "End" "Algebras" ";"
              (!1 "Focus" !1 focus "End" "Focus" ";")
              !1 testIntentionInterface? !1 testIntentionBody !1
              "End" ";";
  TestIntentionInterface ::= "Interface" (intention)+;
```

[2] http://www.emftext.org/index.php/EMFText

```
53  IntentionDec ::= "TestIntention" (name[IDENTIFIER] ";");
54  TestIntentionBody ::= "Body" !1 variableDeclaration? !1 axiomDeclaration  ;
55  AxiomDeclaration ::= "Axioms" (!1 axiom ";")+;
56  Axiom ::=  ( condition  "=>" )?  inclusion;
57  Inclusion ::= hmlTerm "in" in[IDENTIFIER];
58  Condition ::= conditionBody;
59  ConditionBody ::= conditionAtom ("," conditionAtom)*;
60
61  SATEL.HMLFormula.HMLTerm ::= hmlFormula ("." hmlFormula)*;
62  SATEL.HMLFormula.HMLFormulaHMLFormulaContent ::= "HML" "(" hmlFormulaContent ")";
63  SATEL.HMLFormula.HMLFormulaPrimitiveHMLVarDec ::= "$" primitiveHMLVarDec[];
64  SATEL.HMLFormula.HMLNext ::= "{" hmlEvent "}" hmlFormulaContent ;
65  SATEL.HMLFormula.HMLNot ::= "not" "("hmlFormulaContent")";
66  SATEL.HMLFormula.HMLAnd ::= "(" hmlFormulaContentL "and" hmlFormulaContentR ")";
67  SATEL.HMLFormula.HMLTop ::= "T";
68  SATEL.HMLFormula.HMLEvent ::= "<" inputTerm ("with" outputTerm )?">";
69  SATEL.HMLFormula.SynchronizationEventInputTerm ::= event[IDENTIFIER]("(" parameters")")?;
70  SATEL.HMLFormula.SynchronizationEventOutputTerm ::=event[IDENTIFIER]("(" parameters")")?;
71  SATEL.HMLFormula.WPrimitiveStimulationVarDec ::= "$" primitiveStimulation[];
72  SATEL.HMLFormula.WPrimitiveObservationVarDec ::= "$" primitiveObservation[];
73
74  SATEL.HMLFormula.Parameter ::=  value ("," next)?;
75
76
77  SATEL.AlgebraicExpressions.algterms.VariableRef ::= "$" var[];
78  SATEL.AlgebraicExpressions.algterms.CompositeTerm  ::=
79                                                  op[] ("^" iter[INTEGER])? ("("terms ("," terms)*")")?;
80  SATEL.AlgebraicExpressions.AlgEquality ::=
81                                                  "{" algebraicTermL "}" "="  "{" algebraicTermR "}";
82  SATEL.AlgebraicExpressions.SyncEquality ::=
83                                                  "<" synchronizationTermL ">" "="  "<" synchronizationTermR ">";
84  SATEL.AlgebraicExpressions.HMLEquality ::= hmlTermL "=" hmlTermR;
85  SATEL.AlgebraicExpressions.BooleanEquality ::= booleanTermL "=" booleanTermR ;
86  SATEL.AlgebraicExpressions.ArithmeticEquality ::= arithmeticTermL "=" arithmeticTermR;
87
88
89  SATEL.AlgebraicExpressions.booleanterms.Not ::= "not" "(" booleanTerm ")";
90  SATEL.AlgebraicExpressions.booleanterms.Sequence ::= "sequence" "("hmlTerm")";
91  SATEL.AlgebraicExpressions.booleanterms.Positive ::= "positive" "("hmlTerm")";
92  SATEL.AlgebraicExpressions.booleanterms.Trace ::= "trace" "("hmlTerm")";
93  SATEL.AlgebraicExpressions.booleanterms.BooleanVariable ::= booleanVariable[IDENTIFIER];
94  SATEL.AlgebraicExpressions.booleanterms.BooleanValue ::= value[] ;
95  SATEL.AlgebraicExpressions.booleanterms.BOPAnd ::="(" booleanTermL "and" booleanTermR ")";
96  SATEL.AlgebraicExpressions.booleanterms.BOPOr ::= "(" booleanTermL "or" booleanTermR")";
97  SATEL.AlgebraicExpressions.booleanterms.Equals ::= arithmeticTermL "==" arithmeticTermR;
98  SATEL.AlgebraicExpressions.booleanterms.NotEquals ::=arithmeticTermR "<>" arithmeticTermL;
99
100 SATEL.AlgebraicExpressions.booleanterms.GT ::=  arithmeticTermL ">" arithmeticTermR;
101 SATEL.AlgebraicExpressions.booleanterms.LT ::=  arithmeticTermL "<"arithmeticTermR ;
102 SATEL.AlgebraicExpressions.booleanterms.GTE ::=  arithmeticTermL ">=" arithmeticTermR  ;
103 SATEL.AlgebraicExpressions.booleanterms.LTE ::=  arithmeticTermL "<=" arithmeticTermR ;
104
105 @Operator(type="primitive", weight="5", superclass="ArithmeticTerm")
106 SATEL.AlgebraicExpressions.arithmeticterms.IntegerVariable ::=  integerVariable[];
```

```
@Operator(type="primitive", weight="5", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.IntegerValue ::= value[INTEGER];
@Operator(type="binary_left_associative", weight="1", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.BOPPlus ::= arithmeticTermL "+" arithmeticTermR;
@Operator(type="binary_left_associative", weight="1", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.BOPMinus ::=arithmeticTermL "-" arithmeticTermR;
@Operator(type="binary_left_associative", weight="4", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.BOPTimes ::=arithmeticTermL "*" arithmeticTermR;
@Operator(type="binary_left_associative", weight="4", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.BOPDiv ::=arithmeticTermL  "/" arithmeticTermR;
@Operator(type="primitive", weight="5", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.NBEvents ::= "nbEvents" "("hmlTerm")";
@Operator(type="primitive", weight="5", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.Depth ::= "depth" "("hmlTerm")";
@Operator(type="primitive", weight="5", superclass="ArithmeticTerm")
SATEL.AlgebraicExpressions.arithmeticterms.UOPMinus ::= "-" "("arithmeticTerm")";


SATEL.VariableDeclarations.VariableDeclaration ::= "Variables"  ( variable ";" )*;
@SuppressWarnings(featureWithoutSyntax)
SATEL.VariableDeclarations.PrimitiveHMLVarDec ::= name[IDENTIFIER] ":" "primitiveHML" ;
@SuppressWarnings(featureWithoutSyntax)
SATEL.VariableDeclarations.PrimitiveStimulationVarDec ::=name[IDENTIFIER] ":" "
    primitiveStimulation";
@SuppressWarnings(featureWithoutSyntax)
SATEL.VariableDeclarations.PrimitiveObservationVarDec ::=name[IDENTIFIER] ":" "
    primitiveObservation";
@SuppressWarnings(featureWithoutSyntax)
SATEL.VariableDeclarations.PrimitiveIntegerVarDec ::=name[IDENTIFIER] ":" "primitiveInteger"
     ;
@SuppressWarnings(featureWithoutSyntax)
SATEL.VariableDeclarations.PrimitiveBooleanVarDec ::=
                                                        name[IDENTIFIER] ":" "
                                                            primitiveBoolean";
@SuppressWarnings(featureWithoutSyntax)
SATEL.VariableDeclarations.AlgebraicSortVarDec ::=   name[IDENTIFIER] ":"   type[] ;

SATEL.APN.apnmm.APN ::= "APN" name[IDENTIFIER]
                ("Places" ownedPlaces+)
                ("Arcs" ownedArcs+)?
                ("Transitions" ownedTransitions+)
                ("Methods" "["methods(","methods*)"]")?
                ("Gates" "["gates(","gates)*"]")?
                ("Where" ownedVariables*)?;
SATEL.APN.apnmm.Arc ::= "Arc"  Name[IDENTIFIER]? "{"
                                    from[] "-->" to[]
                                    ("tokens"  ownedArcMultiset  )?
                                   "}";
SATEL.APN.apnmm.Place ::=  "Place" name[]
                                                    ("sort:" sort)?
                                                    ("tokens"  ownedPlaceMultiset )?;
SATEL.APN.apnmm.Transition ::= "Transition"  name[IDENTIFIER] "{"
                ownedGuard?
                ("GateCalls" "[" (gateCalls)(";"gateCalls)*  "]")?
                ("MethodCalls" "["methodCall "]")? "}";
SATEL.APN.apnmm.Method ::= name[IDENTIFIER];
SATEL.APN.apnmm.Gate ::= name[IDENTIFIER];
```

```
160   SATEL.APN.apnmm.GateCall ::= gate[IDENTIFIER] ("(" params (","params)*")")?;
161   SATEL.APN.apnmm.MethodCall ::=  method[IDENTIFIER] ("(" params ("," params)*")")?;
162   SATEL.APN.guardmm.Guard ::= "Guard" "{"  ownedEquations  ("Where"  ownedVariables*)? "}";
163
164   SATEL.APN.adtmm.ADT ::= "ADT"   name[IDENTIFIER]
165               ("Sort" ownedSorts*)? "Generators" ownedGenerators*
166               ("Operations" ownedOperations* )?
167               ("Axioms" ownedAxioms* )?
168               ("Where" ownedVariables*)? ;
169   SATEL.APN.adtmm.Sort ::= "sort";
170   SATEL.APN.adtmm.Operation ::=  name[] (":" #1 operationSorts ("," operationSorts)*)?
171                                                                    "->" result ";";
172   SATEL.APN.adtmm.Variable ::=   name[IDENTIFIER] ":" variableSort ";";
173   SATEL.APN.adtmm.CondEquation ::= (ownedConditions ("&" ownedConditions)* "=>")?
174                                                          ownedEquation ";" ;
175   SATEL.APN.adtmm.VariableRef ::=  "$" variable[IDENTIFIER];
176   SATEL.APN.adtmm.CTerm ::=   op[] ("^"iter[INTEGER])? ("(" ownedTerms ("," ownedTerms)* ")")?
          ;
177   SATEL.APN.adtmm.Equation ::=   ownedLeftTerm "=" ownedRightTerm ;
178   SATEL.APN.adtmm.Inequation ::= ownedLeftTerm "<>" ownedRightTerm;
179   SATEL.APN.adtmm.AtomicSort ::=  declaration[];
180   SATEL.APN.adtmm.SortDeclaration ::=  name[IDENTIFIER] ;
181
182   SATEL.APN.multisetmm.Multiset ::=  "["(ownedNumOfTerms ( ";" ownedNumOfTerms)*) "]"   ("
          Where" ownedVariables*)?;
183   SATEL.APN.multisetmm.NumOfTerms ::=  ownedElement | ("#"#0 card[INTEGER] "*"  ownedElement);
184   SATEL.APN.multisetmm.TermReference ::= termReferenced;
185 }
```

A.3 Intenções de teste sobre Timer

```
TestIntentionSet mytestintention
    Algebras

        ADT naturals
        Sort nat
        Generators
            zero -> nat ;
            suc : nat -> nat ;
        Operations
            plus : nat, nat -> nat ;
            sum : nat, nat -> nat ;
            eq : nat, nat -> bool;
            lt : nat, nat -> bool ;
        Axioms
            sum($X, zero ) = zero ;
            sum($X, suc($Y))= suc(sum($X,$Y)) ;
            eq(zero, suc($X)) = false;
            eq(suc($Y), zero) = false;
            eq(zero,zero) = true;
            eq(suc($X), suc($Y)) = eq($X,$Y);
            lt (zero, suc($X)) = true;
            lt (suc($X), zero) = false;
            lt (suc($X), suc($Y)) = lt($X,$Y);
        Where

             X : nat ;
             Y : nat    ;

        ADT Boolean
        Sort bool
        Generators
            false -> bool;
            true -> bool;
        Operations
            neg : bool -> bool;
        Axioms
            neg(false) = true ;
            neg(true) = false;
        Where
            X : bool ;

        ADT List
        Sort list
        Generators
            nil -> list ;
            cons : nat ,list -> list;

        Operations
            size : list -> nat ;
            nbOcurr : nat ,list -> nat;

        Axioms
          eq(suc($E),zero)=true => nbOcurr($E, cons($H,$L)) = suc(nbOcurr($E, $L)) ;
          eq($E,$H)=false  => nbOcurr($E, cons($H,$L)) = nbOcurr($E, $L) ;
```

```
55              size (nil) = zero ;
56              size(cons($H,$L)) = suc(size($L)) ;
57              nbOcurr( $E , nil)=zero;
58          Where
59            H: nat ;
60            L : list ;
61            E: nat ;
62      End Algebras ;
63
64      Focus
65          APN Counter
66          Places
67           Place Inc tokens [ zero ]
68           Place Finished
69          Arcs
70           Arc Inc2increment  { Inc --> increment tokens [ $Counter ]  }
71           Arc increment2Inc  {  increment --> Inc tokens  [ suc($Counter)] }
72
73           Arc reset2Inc { reset --> Inc tokens [$Counter]}
74           Arc Inc2reset { Inc --> reset tokens [zero]}
75
76           Arc Inc2alarmTrigger { Inc --> alarmTrigger tokens [$Counter ]}
77           Arc alarmTrigger2Inc { alarmTrigger --> Finished }
78
79          Transitions
80           Transition increment {
81              Guard { lt($Counter, suc^10(zero))=true }
82              MethodCalls [tick]
83            }
84
85           Transition reset {
86              GateCalls [time($Counter) ]
87              MethodCalls [mark]
88            }
89
90            Transition alarmTrigger {
91              Guard { eq($Counter, suc^10(zero))=true }
92              GateCalls [alarm]
93              MethodCalls [tick]
94            }
95
96
97          Methods [tick,mark]
98          Gates [ alarm, time ]
99
100         Where
101             Counter : nat;
102
103     End Focus ;
104
105
106     Interface
107         TestIntention TickIntention;
108         TestIntention MarkIntention;
109         Body
110             Variables
111                 t : primitiveHML    ;
```

```
112                 Counter : nat ;
113                 x : primitiveInteger;
114                 y : primitiveStimulation;
115             Axioms
116                 HML(T) in TickIntention;
117                 $t in TickIntention , nbEvents($t)<6 => $t . HML ( { <tick> } T) in
                        TickIntention;
118                 {lt($Counter ,suc^6(zero))}={false} , $t in TickIntention => $t .HML({<mark
                        with time($Counter)>} T ) in MarkIntention;
119 End
```

A.4 Transformação de refinamento do SATEL

```
-- @atlcompiler atl2006
-- @path SATELAPN=/ATLPreprocessorI/SATELAPN.ecore

module refineSATELAPN;
create OUT : SATELAPN refining IN : SATELAPN;

helper def : setNext(seq: Sequence(OclAny), elt: OclAny) : OclAny=

    elt.refSetValue('next',
                    if elt<> seq.last() then
                                seq.at(seq.indexOf(elt)+1)
                            else
                               OclUndefined
                        endif.debug()
 )
;
rule CTerm  {
    from
        cterm : SATELAPN!CTerm
    to
        out : SATELAPN!CTerm(
                ownedTerms <- cterm.ownedTerms->asSequence()->
                        collect(e | thisModule.setNext(cterm.ownedTerms, e))
        )
}

rule ConditionBody {
    from
        condBody : SATELAPN!ConditionBody
    to
        out : SATELAPN!ConditionBody (
                conditionAtom <- condBody.conditionAtom->asSequence()->
                        collect(e | thisModule.setNext(condBody.conditionAtom, e))
        )
}

rule HMLTerm {
    from
        hmlTerm : SATELAPN!HMLTerm
    to
        out : SATELAPN!HMLTerm(
                hmlFormula  <- hmlTerm.hmlFormula->asSequence()->
                        collect(e | thisModule.setNext(hmlTerm.hmlFormula, e))
        )
}
rule CompositeTerm{
    from
        compositeTerm : SATELAPN!CompositeTerm
    to
        out : SATELAPN!CompositeTerm(
                terms <-    compositeTerm.terms->asSequence()->
                        collect(e | thisModule.setNext(compositeTerm.terms, e))
        )
}
```

```
55  rule Transition{
56      from
57          transition : SATELAPN!Transition
58      to
59          out : SATELAPN!Transition(
60                  gateCalls <-  transition.gateCalls->asSequence()->
61                          collect(e | thisModule.setNext(transition.gateCalls, e))
62          )
63  }
64  rule GateCall{
65      from
66          gateCall : SATELAPN!GateCall
67      to
68          out : SATELAPN!GateCall(
69                  params <-  gateCall.params->asSequence()->
70                          collect(e | thisModule.setNext(gateCall.params, e))
71          )
72  }
73  rule MethodCall{
74      from
75          methodCall : SATELAPN!MethodCall
76      to
77          out : SATELAPN!MethodCall(
78                  params <-  methodCall.params->asSequence()->
79                          collect(e | thisModule.setNext(methodCall.params, e))
80          )
81  }
82  rule Guard{
83      from
84          guard : SATELAPN!Guard
85      to
86          out : SATELAPN!Guard(
87                  ownedEquations <-  guard.ownedEquations->asSequence()->
88                          collect(e | thisModule.setNext(guard.ownedEquations, e))
89          )
90
91  }
92  rule CondEquation{
93      from
94          condEquation : SATELAPN!CondEquation
95      to
96          out : SATELAPN!CondEquation(
97                  ownedConditions <-  condEquation.ownedConditions->asSequence()->
98                          collect(e | thisModule.setNext(condEquation.ownedConditions, e))
99          )
100 }
101 rule Multiset{
102     from
103         multiset: SATELAPN!Multiset
104     to
105         out : SATELAPN!Multiset(
106                 ownedNumOfTerms <- multiset.ownedNumOfTerms->asSequence()->
107                         collect(e | thisModule.setNext(multiset.ownedNumOfTerms, e))
108         )
109 }
```

A.5 Classe Pré-processador

```
1
2  import java.io.IOException;
3  import java.util.Collections;
4  import java.util.Map;
5
6  import org.eclipse.emf.common.util.EList;
7  import org.eclipse.emf.common.util.URI;
8  import org.eclipse.emf.ecore.EObject;
9  import org.eclipse.emf.ecore.resource.Resource;
10 import org.eclipse.emf.ecore.resource.ResourceSet;
11 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
12 import org.eclipse.emf.ecore.util.EcoreUtil;
13 import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
14
15 import SATEL.Model;
16 import SATEL.APN.adtmm.AdtmmFactory;
17 import SATEL.APN.adtmm.CTerm;
18 import SATEL.APN.adtmm.Variable;
19 import SATEL.APN.apnmm.Gate;
20 import SATEL.APN.apnmm.Method;
21 import SATEL.APN.multisetmm.Multiset;
22 import SATEL.APN.multisetmm.NumOfTerms;
23 import SATEL.AlgebraicExpressions.algterms.AlgtermsFactory;
24 import SATEL.AlgebraicExpressions.algterms.CompositeTerm;
25 import SATEL.VariableDeclarations.VariableDec;
26
27
28 /**
29  * Class responsible for (in the order it is presented)
30  * -Load a SATELAPN (original) model
31  * -Replace Variables names for new ones capitalized (as required by prolog) [VariableDec and
        Variable]
32  * -Unfold contracted form of CTerms and CompositeTerms
33  * -Replace gates and methods names [Gate and Methods]
34  * @author RDomingues
35  *
36  */
37 public class PreProcessor  {
38
39     private int varindex ;
40     private int varDecindex;
41
42     public PreProcessor(String inpathfile, String outpathfile){
43         varindex = 0;
44         varDecindex=0;
45         writeSatel(loadSATELApn(inpathfile), outpathfile);
46     }
47
48     private SATEL.Model loadSATELApn(String relativepathfile) {
49         SATEL.SATELPackage.eINSTANCE.eClass();
50         Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
51         Map<String, Object> m = reg.getExtensionToFactoryMap();
52         m.put("SATEL", new XMIResourceFactoryImpl());
53         ResourceSet resSet = new ResourceSetImpl();
```

```
54          Resource resource = resSet.getResource(URI.createURI(relativepathfile), true);
55          visit(resource.getContents().get(0),null,null);
56          return (SATEL.Model) resource.getContents().get(0);
57      }
58
59      private void writeSatel(Model model, String relativepathfile) {
60          Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
61          Map<String, Object> m = reg.getExtensionToFactoryMap();
62          m.put("SATEL", new XMIResourceFactoryImpl());
63          ResourceSet resSet = new ResourceSetImpl();
64          Resource resource = resSet.createResource(URI
65                  .createURI(relativepathfile));
66          resource.getContents().add(model);
67          try {
68              resource.save(Collections.EMPTY_MAP);
69          } catch (IOException e) {
70              e.printStackTrace();
71          }
72      }
73
74      private void visit(EObject eObject, Integer indexOnContainer,  EObject container)  {
75          check(eObject, indexOnContainer, container);
76          EList<EObject> contents = eObject.eContents();
77          for (int i =0; i < contents.size(); i++) {
78              visit(contents.get(i), i, eObject);
79          }
80      }
81
82      private void checkMultiSet(Multiset multiset) {
83          EList<NumOfTerms> notList = multiset.getOwnedNumOfTerms();
84          for (NumOfTerms numOfTerms : notList) {
85              int card = numOfTerms.getCard() ;
86              numOfTerms.setCard(1);
87              if ( card > 1) {
88                  numOfTerms.setCard(1);
89                  for (int i = 0; i < card; i++)
90                      notList.add(EcoreUtil.copy(numOfTerms));
91              }
92          }
93      }
94
95
96      private void checkCTerm(CTerm cterm) {
97          int iter = cterm.getIter(); //save iterations
98          if (iter<=1) return ;
99          cterm.setIter(0);
100         CTerm last = AdtmmFactory.eINSTANCE.createCTerm();
101         last.setIter(0);
102         last.setOp(cterm.getOp());
103         last.getOwnedTerms().addAll(cterm.getOwnedTerms());
104         CTerm lastGenerated = cterm;
105         for (int i = 1; i< iter-1; i++) {
106             CTerm next = AdtmmFactory.eINSTANCE.createCTerm();
107             next.setIter(0);
108             next.setOp(cterm.getOp());
109             lastGenerated.getOwnedTerms().add(next);
110             lastGenerated = next;
```

```
111         }
112         lastGenerated.getOwnedTerms().add(last);
113     }
114     private void checkCompositeTerm(CompositeTerm cterm) {
115         int iter = cterm.getIter();
116         if (iter<=1) return ;
117         cterm.setIter(0);
118         CompositeTerm last = AlgtermsFactory.eINSTANCE.createCompositeTerm();
119         last.setIter(0);
120         last.setOp(cterm.getOp());
121         last.getTerms().addAll(cterm.getTerms());
122         CompositeTerm lastGenerated = cterm;
123         for (int i = 1; i< iter-1; i++) {
124             CompositeTerm next = AlgtermsFactory.eINSTANCE.createCompositeTerm();
125             next.setIter(0);
126             next.setOp(cterm.getOp());
127             lastGenerated.getTerms().add(next);
128             lastGenerated = next;
129         }
130         lastGenerated.getTerms().add(last);
131     }
132 
133     private void check(EObject eObject, Integer indexOnContainer, EObject container) {
134         if (eObject instanceof CompositeTerm) {
135             CompositeTerm casted = (CompositeTerm)eObject;
136             if (casted.getIter()>1) {
137                 System.out.println(casted.getIter());
138                 checkCompositeTerm((CompositeTerm) eObject);
139             }
140         }
141         if (eObject instanceof CTerm) {
142             CTerm casted = (CTerm)eObject;
143             if (casted.getIter()> 1) {
144                 System.out.println(casted.getIter());
145                 checkCTerm(casted);
146             }
147         }else if (eObject instanceof VariableDec) {
148             ((VariableDec) eObject).setName(genVarDec());
149         } else if (eObject instanceof Variable) {
150           String newName = genVar();
151           System.out.println("Var   "+((Variable)eObject).getName() +" passa a ser   "+newName
                );
152             ((Variable) eObject).setName(newName);
153         }
154         else if (eObject instanceof  Gate)
155             ((Gate) eObject).setName("gate_"+((Gate) eObject).getName());
156         else if (eObject instanceof  Method)
157                 ((Method) eObject).setName("meth_"+((Method) eObject).getName());
158         else if (eObject instanceof Multiset)
159                 checkMultiSet((Multiset) eObject);
160     }
161 
162     private String genVar() {
163         return "V"+ varindex++;
164     }
165     private String genVarDec() {
166         return "D"+ varDecindex++;
```

```
167     }
168
169     public static void main(String ...strings) {
170         new PreProcessor(strings[0], strings[1]);
171         System.out.println("Preprocessing... done.");
172     }
173 }
```

A.6 Classe Pós-processador

```
1
2  import java.io.IOException;
3  import java.util.Collections;
4  import java.util.HashMap;
5  import java.util.Map;
6
7  import mprologTermReference.Model;
8  import mprologTermReference.Variable;
9
10 import org.eclipse.emf.common.util.EList;
11 import org.eclipse.emf.common.util.URI;
12 import org.eclipse.emf.ecore.EObject;
13 import org.eclipse.emf.ecore.resource.Resource;
14 import org.eclipse.emf.ecore.resource.ResourceSet;
15 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
16 import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
17
18 /**
19  * Class responsible for (in the order it is presented)
20  * -Load a SATELAPN (original) model
21  * -Replace Variables names for new ones capitalized (as required by prolog) [VariableDec and
        Variable]
22  * -Unfold contracted form of CTerms and CompositeTerms
23  * -Replace gates and methods names [Gate and Methods]
24  * @author RDomingues
25  *
26  */
27 public class PosProcessor  {
28
29     private HashMap<String, Integer> table;
30     public PosProcessor(String inpathfile, String outpathfile){
31         table = new HashMap<String, Integer>();
32         writeMPrologTR(loadMprologTR(inpathfile), outpathfile);
33     }
34
35     private Model loadMprologTR(String relativepathfile) {
36         mprologTermReference.MprologTermReferencePackage.eINSTANCE.eClass();
37         Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
38         Map<String, Object> m = reg.getExtensionToFactoryMap();
39         m.put("mprologTR", new XMIResourceFactoryImpl());
40         ResourceSet resSet = new ResourceSetImpl();
41         Resource resource = resSet.getResource(URI.createURI(relativepathfile), true);
42         visit(resource.getContents().get(0));
43         return  (Model) resource.getContents().get(0);
44     }
45
46     private void writeMPrologTR(Model model, String relativepathfile) {
47         Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
48         Map<String, Object> m = reg.getExtensionToFactoryMap();
49         m.put("mprologTR", new XMIResourceFactoryImpl());
50         ResourceSet resSet = new ResourceSetImpl();
51         Resource resource = resSet.createResource(URI
52                 .createURI(relativepathfile));
53         resource.getContents().add(model);
```

```
54          try {
55              resource.save(Collections.EMPTY_MAP);
56          } catch (IOException e) {
57              e.printStackTrace();
58          }
59      }
60
61      private void visit(EObject eObject)  {
62          check(eObject);
63          EList<EObject> contents = eObject.eContents();
64          for (EObject contained : contents) {
65              visit(contained);
66          }
67      }
68
69
70      private void check(EObject eObject) {
71          if (eObject instanceof Variable) {
72              Variable v = (Variable)eObject;
73              String vname = v.getName();
74              if (vname.startsWith("##") && vname.startsWith("##") )
75              {
76                  Integer idx = table.get(vname);
77                  idx=idx!=null ?++idx:0;
78                  v.setName("X"+idx);
79                  table.put(vname, idx);
80              }
81          }
82      }
83
84      public static void main(String ...strings) {
85          new PosProcessor(strings[0], strings[1]);
86          System.out.println("Posprocessing... done.");
87      }
88  }
```

A.7 Sintaxe concreta textual do DSLTrans

```
1
2  SYNTAXDEF dsltranstext
3  FOR <http://dsltrans/2.0>
4  START TransformationModel
5
6  OPTIONS{
7      defaultTokenName = "IDENTIFIER";
8      usePredefinedTokens = "false";
9  }
10
11 TOKENS {
12     DEFINE WHITESPACE $(' ' | '\t' | '\f')$;
13     DEFINE LINEBREAK $('\r\n' | '\r' | '\n')$;
14     DEFINE IDENTIFIER $('A'..'Z' | 'a'..'z' | '_')('A'..'Z' | 'a'..'z' | '0'..'9' | '-'| '_' |
           '.')*$;
15     DEFINE INTEGER $('-')?('1'..'9')('0'..'9')*|'0'$;
16     DEFINE FLOAT $('-')?(('1'..'9') ('0'..'9')* | '0') '.' ('0'..'9')+ $;
17     DEFINE SL_COMMENT $'//'(~('\n'|'\r'|'\ufffe'))*$;
18 }
19 TOKENSTYLES {
20      "SL_COMMENT" COLOR #5CB627;
21
22 }
23 RULES {
24     TransformationModel ::=  source* ;
25     FilePort ::= "File" !1
26                  ("id"#1"="#1 name[IDENTIFIER]  !1)?
27                  ("uri"#1"="#1 filePathURI['\'' , '\'' ] )? !1   //path to .xmi model file
28                              metaModelId ;    //identifier
29
30     MetaModelIdentifier ::= "metamodel"#0"(" !2    //specify source of metamodel (Layer or FilePort)
31                              ("mmname" #1 "=" #1 metaModelName[IDENTIFIER] !2)?   //name of the
                                    metamodel
32                              ("uri" #1 "=" #1 metaModelURI['\'' , '\'']? !1")" !0;   //path of
                                    metamodel to its path to .ecore file
33
34     Sequential ::=
35
36         "def"#1(description['\'','\''] #1 ":"#1) ? "layer" #2(name['\'' , '\'']? !1
37         "previous"#1"="#1previousSource['\'','\'']+ !1
38         ("group"#1"="#1groupName[IDENTIFIER] !1)?
39         ("output"#1"="#1 outputFilePathURI['"' , '"' ]!1 )?
40          metaModelId !1
41          (hasRule!1)* !0
42         "end" #1"def"!0 !0;
43
44     Rule ::= "rule" (description['\'' , '\'']? !1
45                 ("match"#1 match)+ !1
46                 ("apply"!2 apply) !1
47                 ("restrictions" !2 (backwards !2)*)? !1
48                 ("import" !2 (imports!2)*)? !0
49               "end"#1"rule" !0 !0; //defines a transformation between input(Match model) and output(Apply model)
                     model
50
```

```
51  MatchModel ::= ("source"#1"="#1 explicitSource[IDENTIFIER] #1)? "with" !2 ( class !2)* !2
        ("subject"#1"to" (!3 association)*)?  ; //holds pattern that we want to capture from onput
        model
52
53  ApplyModel ::=  (class!2)* ("subject" "to" (!3 association)*)? ;
54
55  AnyMatchClass ::= (description[IDENTIFIER]":")? "any" #1 packageName[IDENTIFIER]#0"::"#0
        className[IDENTIFIER]("("    //reason about packageName and
        className
56                  ( #2 attribute)*  #1 ")")? ;
57
58  ExistsMatchClass ::= (description[IDENTIFIER]":")? "existing" #1 packageName[IDENTIFIER]#0
        "::"#0 className[IDENTIFIER] ("("
59                      (#2 attribute*) #1  ")")?  ;
60
61  NegativeMatchClass ::= (description[IDENTIFIER]":")? "not" #1 packageName[IDENTIFIER]#0"::
        "#0 className[IDENTIFIER] ("("
62                      ((#2 attribute)*) #1")")? ;  // NegativeMatchClass
63
64  ApplyClass ::= (description[IDENTIFIER]":")? packageName[IDENTIFIER] #0 "::"#0 className[
        IDENTIFIER] ("("
65              (!3 attribute)* !2 ")")?  #1 ("in" #1"group" groupName['\'' , '\'']])? ;
66
67  PositiveMatchAssociation ::= source[IDENTIFIER] #1"—"#0"(" #0 associationName[IDENTIFIER]
         #0 ")" #0 "–>" #1 target[IDENTIFIER];
68  NegativeMatchAssociation ::= source[IDENTIFIER] #1"!–"#0"(" #0 associationName[IDENTIFIER]
         #0 ")" #0 "–>" #1 target[IDENTIFIER] ;
69  PositiveIndirectAssociation ::= source[IDENTIFIER] #1"~~"#0"(" #0 associationName[
        IDENTIFIER] #0 ")" #0 "~>" #1 target[IDENTIFIER] ;
70  NegativeIndirectAssociation ::= source[IDENTIFIER] #1"!~"#0"(" #0 associationName[
        IDENTIFIER] #0 ")" #0 "~>" #1 target[IDENTIFIER] ;
71
72  ApplyAssociation ::= source[IDENTIFIER]"—"#0"(" #0 associationName[IDENTIFIER] #0 ")" #0
        "–>" #1 target[IDENTIFIER];
73
74  MatchAttribute ::=  (description[IDENTIFIER ]#1":"#1)?  attributeName[IDENTIFIER] ("="
        attributeValue)?  ; //
        MatchAttribute
75
76  ApplyAttribute ::=  ((description[IDENTIFIER ]#1":"#1)? attributeName[IDENTIFIER] ("="#1
        attributeValue)?) | ("self" #1 "=" #1 attributeValue)   ;
77
78
79  PositiveBackwardRestriction  ::= targetClass[IDENTIFIER] #1 "derived" #1 "from" #1
        sourceClass[IDENTIFIER] ;
80
81  NegativeBackwardRestriction ::= targetClass[IDENTIFIER] #1 "not" #1 "derived" "from" #1
        sourceClass[IDENTIFIER] ;
82
83  Import ::=  "("#1 "target" #1 "=" target[IDENTIFIER]#0","#1 "source" #1"="#1  source[
        IDENTIFIER] ")"; //Import
84
85  Atom ::=  value['\'','\''] ; //Atom
86
87  AttributeRef ::= "sameAs"#0"(" attributeRef[IDENTIFIER] ")" ;
88
89  ClassRef ::= "sameAs"#0"(" classRef[IDENTIFIER]")" ; //("id:"description[])?
        ":"packageName[]"."className[] ";"; //ClassRef
90
```

```
91      Concat ::= "concat" #0 "(" leftTerm "with" rightTerm ")";    //Concat
92
93      TypeOf ::= "typeOf" attributeRef[] ;//TypeOf
94
95      Wildcard ::= "WILDCARD";
96
97      isNull ::= "isNull"#0"="value["true" : "false" ]   ; //isNull
98
99  }
```

A.8 Modelo de Transformação

```
File
    id = input
    uri = 'timermodel.SATEL'
    metamodel(
                mmname = SATEL.SATEL
                uri = 'SATELAPN.ecore'
            )

def 'Direct Mappings' : layer  'Entities'
    previous = 'input'
    output = "result.mprologTR"
    metamodel(
            mmname = mprologTermReference.MprologTermReference
            uri = 'mprologTermReference.ecore'
        )

    rule 'SatelModel2MPrologModel'
        match with
            any SATEL::Model
        apply
                mprologTermReference::Model(
                        self = 'Model'
                        name= 'model'
                    )
    end rule


    rule 'CondEquation2ClauseWith_Head_Body'
        match with
            m2: any SATEL.APN.adtmm::CondEquation
        apply
            m3: mprologTermReference::Clause(
                        self = 'CondEquationClause'
                    )
            m4:     mprologTermReference::Head(
                        self = 'CondEquationClauseHead'
                    )
            m5:
                    mprologTermReference::Body(
                        self = 'CondEquationClauseBody'
                    )
             subject to
                m3 --(ownedHead)-> m4
                m3 --(ownedBody)-> m5
    end rule

    rule 'CTerm2Functor'
        match with
            m6: any SATEL.APN.adtmm::CTerm
            m7: any SATEL.APN.adtmm::Operation(  a8 :  name )
             subject to
                m6 --(op)-> m7
        apply
            m9: mprologTermReference::Functor(
```

```
55                      text= sameAs(a8)
56                      self = 'TermFunctor'
57                  )
58      end rule
59
60      rule 'Equation2Left_and_Right_EvalFunctor_and_Variables for Result'
61          match with
62              m10: any SATEL.APN.adtmm::Equation
63          apply
64              m11: mprologTermReference::Functor(
65                      text= 'eval'
66                      self = 'LeftEvalFunctor'
67                  )
68              m12: mprologTermReference::Functor(
69                      text= 'eval'
70                      self = 'RightEvalFunctor'
71                  )
72              m13: mprologTermReference::Variable(
73                      name= '##var##'
74                      self = 'EvalFunctorVariable'
75                  )
76              m14: mprologTermReference::VariableReference(
77                      self = 'EvalFunctorVariableReference'
78                  )
79              subject to
80                  m14 --(idReference)-> m13
81      end rule
82
83      rule 'Generator2Clause'
84          match with
85              m15: any SATEL.APN.adtmm::ADT
86              m16: any SATEL.APN.adtmm::Operation
87              subject to
88                  m15 --(ownedGenerators)-> m16
89          apply
90              m17: mprologTermReference::Clause(
91                      self = 'GeneratorClause'
92                  )
93      end rule
94
95
96      rule 'Generator2Head'
97          match with
98              m18: any SATEL.APN.adtmm::ADT
99              m19: any SATEL.APN.adtmm::Operation
100             subject to
101                 m18 --(ownedGenerators)-> m19
102         apply
103             m20: mprologTermReference::Head(
104                     self = 'GeneratorHead'
105                 )
106     end rule
107
108
109     rule 'Generator Body'
110         match with
111             m21: any SATEL.APN.adtmm::ADT
```

```
112             m22: any SATEL.APN.adtmm::Operation
113
114             subject to
115                 m21 --(ownedGenerators)-> m22
116         apply
117             m23: mprologTermReference::Body(
118                         self = 'GeneratorBody'
119                     )
120     end rule
121
122
123     rule 'OperationNameAndResultAtomicSort2HeadFunctors'
124         match with
125             m24: any SATEL.APN.adtmm::Operation(  a25 :  name )
126             m26: any SATEL.APN.adtmm::AtomicSort
127             m27: any SATEL.APN.adtmm::SortDeclaration(  a28 :  name )
128             m29: any SATEL.APN.adtmm::ADT
129
130             subject to
131                 m24 --(result)-> m26
132                 m26 --(declaration)-> m27
133                 m29 --(ownedGenerators)-> m24
134         apply
135             m30: mprologTermReference::Functor(
136                         self = 'OperationFunctor'
137                         text= sameAs(a25)
138                     )
139             m31: mprologTermReference::Functor(
140                         self = 'ResultFunctor'
141                         text= sameAs(a28)
142                     )
143             subject to
144                 m31 --(ownedTerm)-> m30
145     end rule
146
147     rule 'Foreach_OperationSort_in_Operation2HeadVariablesAndBodyFunctors'
148         match with
149             m32: any SATEL.APN.adtmm::Operation(  a33 :  name )
150             m34: any SATEL.APN.adtmm::AtomicSort
151             m35: any SATEL.APN.adtmm::SortDeclaration(  a36 :  name )
152             m37: any SATEL.APN.adtmm::ADT
153             subject to
154                 m32 --(operationSorts)-> m34
155                 m34 --(declaration)-> m35
156                 m37 --(ownedGenerators)-> m32
157         apply
158             m38: mprologTermReference::Variable(
159                         self = 'AtomicSortVariable'
160                         name= '##var##'
161                     )
162             m39: mprologTermReference::Functor(
163                         self = 'OperationSortFunctor'
164                         text= sameAs(a36)
165                     )
166             m40:
167                     mprologTermReference::VariableReference(
168                         self = 'AtomicSortVariableRef'
```

```
169                     )
170             subject to
171                 m40 --(idReference)-> m38
172                 m39 --(ownedTerm)-> m40
173     end rule
174
175     rule 'Generator2EvalClause'
176         match with
177             m41: any SATEL.APN.adtmm::ADT
178             m42: any SATEL.APN.adtmm::Operation
179             subject to
180                 m41 --(ownedGenerators)-> m42
181         apply
182             m43:mprologTermReference::Clause(
183                         self = 'GeneratorClauseEval'
184                     )
185     end rule
186
187     rule 'Generator2HeadWithArgAndResultEvalFunctor'
188         match with
189             m44: any SATEL.APN.adtmm::ADT
190             m45: any SATEL.APN.adtmm::Operation(  a46 :  name )
191             subject to
192                 m44 --(ownedGenerators)-> m45
193         apply
194             m47: mprologTermReference::Head(
195                         self = 'GeneratorHeadEval'
196                     )
197             m48: mprologTermReference::Functor(
198                         text= 'eval'
199                     )
200             m49: mprologTermReference::Functor(
201                         text= sameAs(a46)
202                         self = 'GeneratorEvalFirst'
203                     )
204             m50: mprologTermReference::Functor(
205                         text= sameAs(a46)
206                         self = 'GeneratorEvalSecond'
207                     )
208             subject to
209                 m47 --(ownedFunctor)-> m48
210                 m49 --(nextTerm)-> m50
211                 m48 --(ownedTerm)-> m49
212     end rule
213
214     rule 'Generator2EvalBody'
215         match with
216             m51: any SATEL.APN.adtmm::ADT
217             m52: any SATEL.APN.adtmm::Operation
218
219             subject to
220                 m51 --(ownedGenerators)-> m52
221         apply
222             m53: mprologTermReference::Body(
223                         self = 'GeneratorBodyEval'
224                     )
225     end rule
```

```
226
227     rule 'OperationSort2SortEvalFunctorWithTwoVarReferencesUsedInTheHead'
228         match with
229             m54: any SATEL.APN.adtmm::Operation
230             m55: any SATEL.APN.adtmm::AtomicSort
231             m56: any SATEL.APN.adtmm::ADT
232
233             subject to
234                 m54 --(operationSorts)-> m55
235                 m56 --(ownedGenerators)-> m54
236         apply
237             m57: mprologTermReference::Variable(
238                         self = 'SortVar'
239                         name= '##var##'
240                     )
241             m58: mprologTermReference::VariableReference
242             m59: mprologTermReference::Variable(
243                         self = 'SortVarAfterEval'
244                         name= '##var##'
245                     )
246             m60: mprologTermReference::VariableReference
247             m61: mprologTermReference::Functor(
248                         self = 'SortEvalFunctor'
249                         text= 'eval'
250                     )
251             subject to
252                 m58 --(idReference)-> m57
253                 m60 --(idReference)-> m59
254                 m61 --(ownedTerm)-> m58
255                 m58 --(nextTerm)-> m60
256     end rule
257
258     rule 'Axiom2ClauseWithHead'
259         match with
260             m62:
261                     any SATEL::Axiom
262
263         apply
264             m63: mprologTermReference::Clause(
265                         self = 'AxiomClause'
266                     )
267             m64: mprologTermReference::Head(
268                         self = 'AxiomClauseHead'
269                     )
270             subject to
271                 m63 --(ownedHead)-> m64
272     end rule
273
274     rule 'booleanNot2Functor'
275         match with
276             m65: any SATEL.AlgebraicExpressions.booleanterms::Not
277         apply
278             m66: mprologTermReference::Functor(
279                         self = 'CAtomFunctor'
280                         text= 'boolNot'
281                     )
282     end rule
```

```
283
284     rule 'booleanSequence2Functor'
285         match with
286             m67:
287                     any SATEL.AlgebraicExpressions.booleanterms::Sequence
288
289         apply
290             m68: mprologTermReference::Functor(
291                         self = 'CAtomFunctor'
292                         text= 'boolSequence'
293                     )
294     end rule
295
296     rule 'booleanPositive2Functor'
297         match with
298             m69: any SATEL.AlgebraicExpressions.booleanterms::Positive
299         apply
300             m70: mprologTermReference::Functor(
301                         self = 'CAtomFunctor'
302                         text= 'boolPositive'
303                     )
304     end rule
305
306
307     rule 'booleanTrace2Functor'
308         match with
309             m71: any SATEL.AlgebraicExpressions.booleanterms::Trace
310         apply
311             m72: mprologTermReference::Functor(
312                         self = 'CAtomFunctor'
313                         text= 'boolTrace'
314                     )
315     end rule
316
317     rule 'BooleanVariable'
318         match with
319             m73: any SATEL.AlgebraicExpressions.booleanterms::BooleanVariable
320             m74: any SATEL.VariableDeclarations::PrimitiveBooleanVarDec( a75 : name )
321
322             subject to
323                 m73 --(booleanVariable)-> m74
324
325         apply
326             m76: mprologTermReference::Functor(
327                         self = 'CAtomFunctor'
328                         text= 'variable'
329                     )
330             m77: mprologTermReference::Variable(
331                         name= sameAs(a75)
332                     ) in group''
333             subject to
334                 m76 --(ownedTerm)-> m77
335     end rule
336
337     rule 'BooleanValue'
338         match with
339             m78: any SATEL.AlgebraicExpressions.booleanterms::BooleanValue( a79 : value )
```

```
340         apply
341             m80: mprologTermReference::Functor(
342                         self = 'CAtomFunctor'
343                         text= 'value'
344                     )
345             m81:
346                     mprologTermReference::Functor(
347                         text= sameAs(a79)
348                     )
349             subject to
350                 m80 --(ownedTerm)-> m81
351     end rule
352
353     rule 'BooleanAnd'
354         match with
355             m82:
356                     any SATEL.AlgebraicExpressions.booleanterms::BOPAnd
357         apply
358             m83:
359                     mprologTermReference::Functor(
360                         self = 'CAtomFunctor'
361                         text= 'boolAnd'
362                     )
363     end rule
364
365     rule 'BooleanOr'
366         match with
367             m84: any SATEL.AlgebraicExpressions.booleanterms::BOPOr
368         apply
369             m85: mprologTermReference::Functor(
370                         self = 'CAtomFunctor'
371                         text= 'boolOr'
372                     )
373     end rule
374
375     rule 'BooleanEquals'
376         match with
377             m86: any SATEL.AlgebraicExpressions.booleanterms::Equals
378         apply
379             m87: mprologTermReference::Functor(
380                         self = 'CAtomFunctor'
381                         text= 'boolEquals'
382                     )
383     end rule
384
385     rule 'BooleanNotEquals'
386         match with
387             m88: any SATEL.AlgebraicExpressions.booleanterms::NotEquals
388         apply
389             m89: mprologTermReference::Functor(
390                         self = 'CAtomFunctor'
391                         text= 'boolNotEquals'
392                     )
393     end rule
394
395     rule 'BooleanGT'
396         match with
```

```
397             m90: any SATEL.AlgebraicExpressions.booleanterms::GT
398         apply
399             m91: mprologTermReference::Functor(
400                         self = 'CAtomFunctor'
401                         text= 'boolGT'
402                     )
403     end rule
404
405     rule 'BooleanLT'
406         match with
407             m92: any SATEL.AlgebraicExpressions.booleanterms::LT
408         apply
409             m93:mprologTermReference::Functor(
410                         self = 'CAtomFunctor'
411                         text= 'boolLT'
412                     )
413     end rule
414
415     rule 'BooleanLTE'
416         match with
417             m94:
418                     any SATEL.AlgebraicExpressions.booleanterms::LTE
419         apply
420             m95: mprologTermReference::Functor(
421                         self = 'CAtomFunctor'
422                         text= 'boolLTE'
423                     )
424     end rule
425
426     rule 'BooleanGTE'
427         match with
428             m96: any SATEL.AlgebraicExpressions.booleanterms::GTE
429         apply
430             m97: mprologTermReference::Functor(
431                         self = 'CAtomFunctor'
432                         text= 'boolGTE'
433                     )
434     end rule
435
436     rule 'IntegerVariable'
437         match with
438             m98: any SATEL.AlgebraicExpressions.arithmeticterms::IntegerVariable
439             m99: any SATEL.VariableDeclarations::PrimitiveIntegerVarDec( a100 : name )
440             subject to
441                 m98 --(integerVariable)-> m99
442         apply
443             m101: mprologTermReference::Variable(
444                         name= sameAs(a100)
445                         self = 'ArithmeticFunctor'
446                     )
447     end rule
448
449     rule 'IntegerValue'
450         match with
451             m102: any SATEL.AlgebraicExpressions.arithmeticterms::IntegerValue( a103 : value
                    )
452         apply
```

```
453             m104: mprologTermReference::Functor(
454                         text= sameAs(a103)
455                         self = 'ArithmeticFunctor'
456                     )
457     end rule
458
459     rule 'ArithmeticPlus'
460         match with
461             m105: any SATEL.AlgebraicExpressions.arithmeticterms::BOPPlus
462         apply
463             m106: mprologTermReference::Functor(
464                         self = 'ArithmeticFunctor'
465
466                         text= 'bopPlus'
467                     )
468     end rule
469
470     rule 'ArithmeticMinus'
471         match with
472             m107:
473                     any SATEL.AlgebraicExpressions.arithmeticterms::BOPMinus
474         apply
475             m108:
476                     mprologTermReference::Functor(
477                         self = 'ArithmeticFunctor'
478                         text= 'bopMinus'
479                     )
480     end rule
481
482     rule 'ArithmeticTimes'
483         match with
484             m109: any SATEL.AlgebraicExpressions.arithmeticterms::BOPTimes
485         apply
486             m110: mprologTermReference::Functor(
487                         self = 'ArithmeticFunctor'
488                         text= 'bopTimes'
489                     )
490     end rule
491
492     rule 'ArithmeticDiv'
493         match with
494             m111: any SATEL.AlgebraicExpressions.arithmeticterms::BOPDiv
495         apply
496             m112: mprologTermReference::Functor(
497                         self = 'ArithmeticFunctor'
498                         text= 'bopDiv'
499                     )
500     end rule
501
502     rule 'ArithmeticNBEvents'
503         match with
504             m113: any SATEL.AlgebraicExpressions.arithmeticterms::NBEvents
505         apply
506             m114: mprologTermReference::Functor(
507                         self = 'ArithmeticFunctor'
508                         text= 'nbEvents'
509                     )
```

```
510     end rule
511
512     rule 'ArithmeticDepth'
513         match with
514             m115: any SATEL.AlgebraicExpressions.arithmeticterms::Depth
515         apply
516             m116: mprologTermReference::Functor(
517                         self = 'ArithmeticFunctor'
518                         text= 'depth'
519                     )
520     end rule
521
522
523     rule 'UOPMinus'
524         match with
525             m117:
526                     any SATEL.AlgebraicExpressions.arithmeticterms::UOPMinus
527         apply
528             m118:
529                     mprologTermReference::Functor(
530                         self = 'ArithmeticFunctor'
531                         text= 'uopMinus'
532                     )
533     end rule
534
535
536     rule 'AlgEquality'
537         match with
538             m119: any SATEL.AlgebraicExpressions::AlgEquality
539         apply
540             m120: mprologTermReference::Functor(
541                         self = 'CAtomFunctor'
542                         text= 'algEquality'
543                     )
544     end rule
545
546     rule 'SyncEquality'
547         match with
548             m121: any SATEL.AlgebraicExpressions::SyncEquality
549         apply
550             m122: mprologTermReference::Functor(
551                         self = 'CAtomFunctor'
552                         text= 'syncEquality'
553                     )
554     end rule
555
556     rule 'HmlEquality'
557         match with
558             m123: any SATEL.AlgebraicExpressions::HMLEquality
559         apply
560             m124: mprologTermReference::Functor(
561                         self = 'CAtomFunctor'
562                         text= 'hmlEquality'
563                     )
564     end rule
565
566     rule 'BooleanEquality'
```

```
567         match with
568             m125: any SATEL.AlgebraicExpressions::BooleanEquality
569         apply
570             m126: mprologTermReference::Functor(
571                         self = 'CAtomFunctor'
572                         text= 'booleanEquality'
573                     )
574     end rule
575
576
577     rule 'arithmeticEquality'
578         match with
579             m127: any SATEL.AlgebraicExpressions::ArithmeticEquality
580         apply
581             m128: mprologTermReference::Functor(
582                         self = 'CAtomFunctor'
583                         text= 'arithmeticEquality'
584                     )
585     end rule
586
587     rule 'AlgebraicVariableRef2Variable'
588         match with
589             m129: any SATEL.AlgebraicExpressions.algterms::VariableRef
590             m130: any SATEL.VariableDeclarations::AlgebraicSortVarDec( a131 : name )
591             subject to
592                 m129 --(var)-> m130
593         apply
594             m132: mprologTermReference::Variable(
595                         name= sameAs(a131)
596                         self = 'AlgTermFunctor'
597                     )
598     end rule
599
600     rule 'CompositeTerm2Functor'
601         match with
602             m133: any SATEL.AlgebraicExpressions.algterms::CompositeTerm
603             m134: any SATEL.APN.adtmm::Operation( a135 : name )
604             subject to
605                 m133 --(op)-> m134
606         apply
607             m136: mprologTermReference::Functor(
608                         text= sameAs(a135)
609                         self = 'AlgTermFunctor'
610                     )
611     end rule
612
613     rule 'ConditionAtom2EvalFunctor'
614         match with
615             m137: any SATEL::ConditionBody
616             m138: any SATEL::ConditionAtom
617             subject to
618                 m137 --(conditionAtom)-> m138
619         apply
620             m139: mprologTermReference::Functor(
621                         text= 'eval'
622                         self = 'CondAtomEvalFunctor'
623                     )
```

```
624     end rule
625
626     rule 'ConditionBody2AxiomBody'
627         match with
628             m140: any SATEL::ConditionBody
629         apply
630             m141: mprologTermReference::Body(
631                         self = 'CondBodyFunctor'
632                     )
633     end rule
634
635     rule 'Inclusion2Functor'
636         match with
637             m142: any SATEL::Inclusion
638         apply
639             m143: mprologTermReference::Functor(
640                         self = 'CAtomFunctor'
641                         text= 'in'
642                     )
643     end rule
644
645     rule 'HMLNextInsideHMLFormContent2Functor'
646         match with
647             m144: any SATEL.HMLFormula::HMLFormulaHMLFormulaContent
648             m145: any SATEL.HMLFormula::HMLNext
649             subject to
650                 m144 --(hmlFormulaContent)-> m145
651         apply
652             m146:
653                     mprologTermReference::Functor(
654                         self = 'hmlFormulaFunctor'
655                         text= 'next'
656                     )
657     end rule
658
659     rule 'HMLNextHMLEvent'
660         match with
661             m147: any SATEL.HMLFormula::HMLEvent
662         apply
663             m148:mprologTermReference::Functor(
664                         self = 'eventFunctor'
665                         text= 'event'
666                     )
667     end rule
668
669     rule 'Inequation2InfixExpressionTwoResultVariablesAndEvalFunctor4BothMembers'
670         match with
671             m149: any SATEL.APN.adtmm::Inequation
672         apply
673             m150: mprologTermReference::Functor(
674                         text= 'eval'
675                         self = 'LeftEvalFunctor'
676                     )
677             m151: mprologTermReference::Functor(
678                         text= 'eval'
679                         self = 'RightEvalFunctor'
680                     )
```

```
681             m152: mprologTermReference::Variable(
682                         name= '##var##'
683                         self = 'REvalFunctorVariable'
684                     )
685             m153: mprologTermReference::VariableReference(
686                         self = 'REvalFunctorVariableRef'
687                     )
688             m154: mprologTermReference::InfixExpression(
689                         self = 'IneqInfixExp'
690                     )
691             m155: mprologTermReference::Operator(
692                         symbol= '\='
693                     )
694             m156: mprologTermReference::VariableReference
695             m157: mprologTermReference::Variable(
696                         name= '##var##'
697                         self = 'EvalFunctorVariable'
698                     )
699             subject to
700                 m153 --(idReference)-> m152
701                 m151 --(nextTerm)-> m154
702                 m154 --(ownedOperator)-> m155
703                 m154 --(rightTerm)-> m153
704                 m154 --(leftTerm)-> m156
705                 m156 --(idReference)-> m157
706     end rule
707
708     rule 'SynchEventInputTerm2Functor'
709         match with
710             m158: any SATEL.HMLFormula::SynchronizationEventInputTerm
711             m159: any SATEL.APN.apnmm::Method( a160 : name )
712             subject to
713                 m158 --(event)-> m159
714         apply
715             m161: mprologTermReference::Functor(
716                         self = 'syncTerm'
717                         text= sameAs(a160)
718                     )
719     end rule
720
721     rule 'SynchronizationEventOutputTerm2Functor'
722         match with
723             m162: any SATEL.HMLFormula::SynchronizationEventOutputTerm
724             m163: any SATEL.APN.apnmm::Gate( a164 : name )
725             subject to
726                 m162 --(event)-> m163
727         apply
728             m165:
729                     mprologTermReference::Functor(
730                         self = 'syncTerm'
731                         text= sameAs(a164)
732                     )
733     end rule
734
735     rule 'WPrimitiveObservationVarDec2Functor'
736         match with
737             m166: any SATEL.HMLFormula::WPrimitiveObservationVarDec
```

```
738             m167: any SATEL.VariableDeclarations::PrimitiveObservationVarDec(  a168 :  name )
739             subject to
740                 m166 --(primitiveObservation)-> m167
741         apply
742             m169:
743                     mprologTermReference::Variable(
744                         self = 'syncTerm'
745                         name= sameAs(a168)
746                     )
747     end rule
748
749     rule 'HMLTerm2List'
750         match with
751             m170: any SATEL.HMLFormula::HMLTerm
752         apply
753             m171: mprologTermReference::List(
754                         self = 'hmlTermList'
755                     )
756     end rule
757
758     rule 'IntentionDecInInclustion2QuotedAtom'
759         match with
760             m172: any SATEL::IntentionDec(  a173 :  name )
761             m174: any SATEL::Inclusion
762             subject to
763                 m174 --(_in)-> m172
764
765         apply
766             m175: mprologTermReference::QuotedAtom(
767                         text= sameAs(a173)
768                         ApplyAttribute= 'intDecQA'
769                     )
770     end rule
771
772     rule 'WPrimitiveStimulationVarDec2Functor'
773         match with
774             m176: any SATEL.HMLFormula::WPrimitiveStimulationVarDec
775             m177: any SATEL.VariableDeclarations::PrimitiveStimulationVarDec(  a178 :  name )
776             subject to
777                 m176 --(primitiveStimulation)-> m177
778
779         apply
780             m179:
781                     mprologTermReference::Variable(
782                         self = 'syncTerm'
783                         name= sameAs(a178)
784                     )
785     end rule
786
787
788     rule 'HMLTopInsideHMLFormContent2Functor'
789         match with
790             m180: any SATEL.HMLFormula::HMLFormulaHMLFormulaContent
791             m181: any SATEL.HMLFormula::HMLTop
792             subject to
793                 m180 --(hmlFormulaContent)-> m181
794         apply
```

```
795             m182: mprologTermReference::Functor(
796                         self = 'hmlFormulaFunctor'
797                         text= 'top'
798                     )
799         end rule
800
801     rule 'HMLFormulaPrimitiveHMLVarDec2Functor'
802         match with
803             m183: any SATEL.HMLFormula::HMLFormulaPrimitiveHMLVarDec
804             m184: any SATEL.VariableDeclarations::PrimitiveHMLVarDec( a185 : name )
805             subject to
806                 m183 --(primitiveHMLVarDec)-> m184
807         apply
808             m186:
809                     mprologTermReference::Variable(
810                         self = 'hmlFormulaFunctor'
811                         name= sameAs(a185)
812                     )
813     end rule
814
815     rule 'HMLNextNotInsideHMLFormulaHMLFormulaContent'
816         match with
817             m187: any SATEL.HMLFormula::HMLNext
818             m188: not SATEL.HMLFormula::HMLFormulaHMLFormulaContent
819             subject to
820                 m188 !-(hmlFormulaContent)-> m187
821         apply
822             m189: mprologTermReference::Functor(
823                         self = 'hmlFormulaFunctor'
824                         text= 'next'
825                     )
826     end rule
827
828     rule 'HMLTopNotInsideHMLFormContent2Functor'
829         match with
830             m190: any SATEL.HMLFormula::HMLTop
831             m191: not SATEL.HMLFormula::HMLFormulaHMLFormulaContent
832             subject to
833                 m191 !-(hmlFormulaContent)-> m190
834         apply
835             m192: mprologTermReference::Functor(
836                         self = 'hmlFormulaFunctor'
837                         text= 'top'
838                     )
839     end rule
840
841     rule 'VariableRef2Variable'
842         match with
843             m193: any SATEL.APN.adtmm::VariableRef
844             m194: any SATEL.APN.adtmm::Variable( a195 : name )
845             subject to
846                 m193 --(variable)-> m194
847         apply
848             m196: mprologTermReference::Variable(
849                         name= sameAs(a195)
850                         self = 'TermFunctor'
851                     )
```

```
852     end rule
853
854     rule 'APN'
855         match with
856             m197: any SATEL.APN.apnmm::APN
857         apply
858             m198: mprologTermReference::Functor(
859                         ApplyAttribute= 'InitialMarcationFunctor'
860                         text= 'initialMarcation'
861                     )
862             m199: mprologTermReference::List(
863                         ApplyAttribute= 'InitialMarcationList'
864                     )
865             subject to
866                 m198 --(ownedTerm)-> m199
867     end rule
868
869     rule 'Multiset'
870         match with
871             m200: any SATEL.APN.multisetmm::Multiset
872         apply
873             m201: mprologTermReference::List(
874                         self = 'MSetList'
875                     )
876     end rule
877
878     rule 'Place'
879         match with
880             m202: any SATEL.APN.apnmm::Place( a203 : name )
881         apply
882             m204: mprologTermReference::Functor(
883                         self = 'PlaceFunctor'
884                         text= 'place'
885                     )
886             m205: mprologTermReference::QuotedAtom(
887                         self = 'PlaceOwnedTerm'
888                         text= sameAs(a203)
889                     )
890             subject to
891                 m204 --(ownedTerm)-> m205
892     end rule
893
894
895     rule 'APN'
896         match with
897             m206: any SATEL.APN.apnmm::Transition
898         apply
899             m207: mprologTermReference::Functor(
900                         ApplyAttribute= 'TransitionFunctor'
901                         text= 'transition'
902                     )
903             m208: mprologTermReference::List(
904                         ApplyAttribute= 'inMethods'
905                     )
906             subject to
907                 m207 --(ownedTerm)-> m208
908     end rule
```

```
909 end def
910
911 def 'Direct Mappings' : layer  'Entities'
912     previous = 'Entities'
913     output = "result.mprologTR"
914     metamodel(
915             mmname = mprologTermReference.MprologTermReference
916              uri = 'mprologTermReference.ecore'
917         )
918
919     rule 'CondEquationClause inside Model'
920         match with
921             m209: any SATEL::Model
922             m210: any SATEL.APN.adtmm::CondEquation
923         apply
924             m211: mprologTermReference::Model(
925                         self = 'Model'
926                     )
927             m212: mprologTermReference::Clause(
928                         self = 'CondEquationClause'
929                     )
930             subject to
931                 m211 --(ownedClause)-> m212
932         restrictions
933             m211 derived from m209
934             m212 derived from m210
935     end rule
936
937     rule 'CondEquation OwnedEquation LeftTerm inside ClauseHead'
938         match with
939             m213: any SATEL.APN.adtmm::CondEquation
940             m214: any SATEL.APN.adtmm::Equation
941             subject to
942                 m213 --(ownedEquation)-> m214
943         apply
944             m215: mprologTermReference::Head(
945                         self = 'CondEquationClauseHead'
946                     )
947             m216: mprologTermReference::Functor(
948                         self = 'LeftEvalFunctor'
949                     )
950             subject to
951                 m215 --(ownedFunctor)-> m216
952         restrictions
953             m215 derived from m213
954             m216 derived from m214
955     end rule
956
957 rule 'AbstractEquationLeftTerm'
958         match with
959             m217: any SATEL.APN.adtmm::AbstractEquation
960             m218: any SATEL.APN.adtmm::Term
961             subject to
962                 m217 --(ownedLeftTerm)-> m218
963         apply
964             m219: mprologTermReference::Functor(
965                         self = 'LeftEvalFunctor'
```

```
966                     )
967             m220: mprologTermReference::Term(
968                         self = 'TermFunctor'
969                     )
970             m221: mprologTermReference::Variable(
971                         self = 'EvalFunctorVariable'
972                     )
973             subject to
974                 m219 -(ownedTerm)-> m220
975                 m220 -(nextTerm)-> m221
976         restrictions
977             m219 derived from m217
978             m220 derived from m218
979             m221 derived from m217
980     end rule
981
982     rule 'EquationRightTerm'
983         match with
984             m222: any SATEL.APN.adtmm::Equation
985             m223: any SATEL.APN.adtmm::Term
986             subject to
987                 m222 -(ownedRightTerm)-> m223
988         apply
989             m224: mprologTermReference::Functor(
990                         self = 'RightEvalFunctor'
991                     )
992             m225: mprologTermReference::Term(
993                         self = 'TermFunctor'
994                     )
995             m226: mprologTermReference::VariableReference(
996                         self = 'EvalFunctorVariableReference'
997                     )
998             subject to
999                 m224 -(ownedTerm)-> m225
1000                m225 -(nextTerm)-> m226
1001        restrictions
1002            m224 derived from m222
1003            m225 derived from m223
1004            m226 derived from m222
1005    end rule
1006
1007    rule 'Term to Term'
1008        match with
1009            m227: any SATEL.APN.adtmm::Term
1010            m228: any SATEL.APN.adtmm::Term
1011
1012            subject to
1013                m227 -(next)-> m228
1014        apply
1015            m229: mprologTermReference::Term(
1016                        self = 'TermFunctor'
1017                    )
1018            m230: mprologTermReference::Term(
1019                        self = 'TermFunctor'
1020                    )
1021            subject to
1022                m230 -(nextTerm)-> m229
```

```
1023
1024        restrictions
1025            m229 derived from m228
1026            m230 derived from m227
1027    end rule
1028
1029    rule 'CtermFirstOwnedTerm'
1030        match with
1031            m231: any SATEL.APN.adtmm::CTerm
1032            m232: any SATEL.APN.adtmm::Term
1033            m233: not SATEL.APN.adtmm::Term
1034            subject to
1035                m233 !-(next)-> m232
1036                m231 --(ownedTerms)-> m232
1037        apply
1038            m234: mprologTermReference::Term(
1039                        self = 'TermFunctor'
1040                    )
1041            m235: mprologTermReference::Term(
1042                        self = 'TermFunctor'
1043                    )
1044            subject to
1045                m235 --(ownedTerm)-> m234
1046
1047        restrictions
1048            m234 derived from m232
1049            m235 derived from m231
1050    end rule
1051
1052    rule 'AbstractEquationInConditions'
1053        match with
1054            m236: any SATEL.APN.adtmm::CondEquation
1055            m237: any SATEL.APN.adtmm::AbstractEquation
1056            subject to
1057                m236 --(ownedConditions)-> m237
1058        apply
1059            m238: mprologTermReference::Functor(
1060                        self = 'RightEvalFunctor'
1061                    )
1062            m239: mprologTermReference::Functor(
1063                        self = 'LeftEvalFunctor'
1064                    )
1065            subject to
1066                m239 --(nextTerm)-> m238
1067        restrictions
1068            m238 derived from m237
1069            m239 derived from m237
1070    end rule
1071
1072    rule 'EquationWithNextAbstractEquation'
1073        match with
1074            m240: any SATEL.APN.adtmm::Equation
1075            m241: any SATEL.APN.adtmm::AbstractEquation
1076            subject to
1077                m240 --(next)-> m241
1078        apply
1079            m242: mprologTermReference::Functor(
```

```
1080                        self = 'LeftEvalFunctor'
1081                    )
1082            m243: mprologTermReference::Functor(
1083                        self = 'RightEvalFunctor'
1084                    )
1085            subject to
1086                m243 --(nextTerm)-> m242
1087        restrictions
1088            m242 derived from m241
1089            m243 derived from m240
1090    end rule
1091
1092    rule 'CondEquationFirstCondition'
1093        match with
1094            m244: any SATEL.APN.adtmm::CondEquation
1095            m245: any SATEL.APN.adtmm::AbstractEquation
1096            m246: not SATEL.APN.adtmm::AbstractEquation
1097            subject to
1098                m244 --(ownedConditions)-> m245
1099                m246 !-(next)-> m245
1100        apply
1101            m247:
1102                    mprologTermReference::Body(
1103                        self = 'CondEquationClauseBody'
1104                    )
1105            m248:
1106                    mprologTermReference::Functor(
1107                        self = 'LeftEvalFunctor'
1108                    )
1109            subject to
1110                m247 --(ownedTerm)-> m248
1111
1112        restrictions
1113            m247 derived from m244
1114            m248 derived from m245
1115    end rule
1116
1117    rule 'AfterLastConditionWeHaveOwnedEquationRightTerm'
1118        match with
1119            m249: any SATEL.APN.adtmm::CondEquation
1120            m250: any SATEL.APN.adtmm::Equation
1121            m251: not SATEL.APN.adtmm::AbstractEquation
1122            m252: any SATEL.APN.adtmm::Equation
1123            subject to
1124                m250 !-(next)-> m251
1125                m249 --(ownedEquation)-> m252
1126                m249 --(ownedConditions)-> m250
1127        apply
1128            m253: mprologTermReference::Functor(
1129                        self = 'RightEvalFunctor'
1130                    )
1131            m254: mprologTermReference::Functor(
1132                        self = 'RightEvalFunctor'
1133                    )
1134            subject to
1135                m253 --(nextTerm)-> m254
1136        restrictions
```

```
1137            m253 derived from m250
1138            m254 derived from m252
1139        end rule
1140
1141        rule 'CondEquationWithNoConditions'
1142            match with
1143                m255: any SATEL.APN.adtmm::CondEquation
1144                m256: not SATEL.APN.adtmm::AbstractEquation
1145                m257: any SATEL.APN.adtmm::Equation
1146                subject to
1147                    m255 !-(ownedConditions)-> m256
1148                    m255 --(ownedEquation)-> m257
1149            apply
1150                m258: mprologTermReference::Body(
1151                            self = 'CondEquationClauseBody'
1152                        )
1153                m259: mprologTermReference::Functor(
1154                            self = 'RightEvalFunctor'
1155                        )
1156                subject to
1157                    m258 --(ownedTerm)-> m259
1158            restrictions
1159                m258 derived from m255
1160                m259 derived from m257
1161        end rule
1162
1163        rule 'Generators inside Model'
1164            match with
1165                m260: any SATEL::Model
1166                m261: any SATEL.APN.adtmm::Operation
1167
1168                subject to
1169                    m260 ~~(contains)~> m261
1170            apply
1171                m262:
1172                        mprologTermReference::Clause(
1173                            self = 'GeneratorClause'
1174                        )
1175                m263:
1176                        mprologTermReference::Model(
1177                            ApplyAttribute= 'Model'
1178                        )
1179                subject to
1180                    m263 --(ownedClause)-> m262
1181
1182            restrictions
1183                m262 derived from m261
1184                m263 derived from m260
1185        end rule
1186
1187        rule 'Generators inside Model'
1188            match with
1189                m264: any SATEL.APN.adtmm::Operation
1190                m265: any SATEL.APN.adtmm::ADT
1191                subject to
1192                    m265 --(ownedGenerators)-> m264
1193            apply
```

```
1194            m266: mprologTermReference::Clause(
1195                        self = 'GeneratorClause'
1196                    )
1197            m267: mprologTermReference::Head(
1198                        self = 'GeneratorHead'
1199                    )
1200            subject to
1201                m266 --(ownedHead)-> m267
1202        restrictions
1203            m266 derived from m264
1204            m267 derived from m264
1205    end rule
1206
1207    rule 'Generators Head.ownedFunctor'
1208        match with
1209            m268: any SATEL.APN.adtmm::Operation
1210            m269: any SATEL.APN.adtmm::AtomicSort
1211            m270: any SATEL.APN.adtmm::ADT
1212            subject to
1213                m268 --(result)-> m269
1214                m270 --(ownedGenerators)-> m268
1215        apply
1216            m271: mprologTermReference::Functor(
1217                        ApplyAttribute= 'ResultFunctor'
1218                    )
1219            m272: mprologTermReference::Head(
1220                        self = 'GeneratorHead'
1221                    )
1222            subject to
1223                m272 --(ownedFunctor)-> m271
1224        restrictions
1225            m271 derived from m269
1226            m272 derived from m268
1227            m271 derived from m268
1228    end rule
1229
1230    rule 'Generator HeadFunctor Variables'
1231        match with
1232            m273: any SATEL.APN.adtmm::Operation
1233            m274: any SATEL.APN.adtmm::AtomicSort
1234            m275: any SATEL.APN.adtmm::ADT
1235            subject to
1236                m273 --(operationSorts)-> m274
1237                m275 --(ownedGenerators)-> m273
1238        apply
1239            m276: mprologTermReference::Variable(
1240                        ApplyAttribute= 'AtomicSortVariable'
1241                    )
1242            m277: mprologTermReference::Functor(
1243                        ApplyAttribute= 'OperationFunctor'
1244                    )
1245            subject to
1246                m277 --(ownedTerm)-> m276
1247                m276 --(nextTerm)-> m276
1248
1249        restrictions
1250            m276 derived from m274
```

```
1251            m277 derived from m273
1252            m276 derived from m275
1253            m276 derived from m273
1254        end rule
1255
1256        rule 'Generators Body Functors'
1257            match with
1258                m278: any SATEL.APN.adtmm::Operation
1259                m279: any SATEL.APN.adtmm::AtomicSort
1260                m280: any SATEL.APN.adtmm::SortDeclaration
1261                m281: any SATEL.APN.adtmm::ADT
1262                subject to
1263                    m278 --(operationSorts)-> m279
1264                    m279 --(declaration)-> m280
1265                    m281 --(ownedGenerators)-> m278
1266            apply
1267                m282: mprologTermReference::Body(
1268                            ApplyAttribute= 'GeneratorBody'
1269                        )
1270                m283: mprologTermReference::Functor(
1271                            ApplyAttribute= 'OperationSortFunctor'
1272                        )
1273                subject to
1274                    m282 --(ownedTerm)-> m283
1275                    m283 --(nextTerm)-> m283
1276            restrictions
1277                m282 derived from m278
1278                m283 derived from m281
1279                m283 derived from m280
1280                m283 derived from m278
1281        end rule
1282
1283        rule 'Creates Body if there is some operationSort'
1284            match with
1285                m284: any SATEL.APN.adtmm::Operation
1286                m285: existing SATEL.APN.adtmm::AtomicSort
1287                m286: any SATEL.APN.adtmm::ADT
1288
1289                subject to
1290                    m284 --(operationSorts)-> m285
1291                    m286 --(ownedGenerators)-> m284
1292
1293            apply
1294                m287: mprologTermReference::Clause(
1295                            self = 'GeneratorClause'
1296                        )
1297                m288: mprologTermReference::Body(
1298                            self = 'GeneratorBody'
1299                        )
1300                subject to
1301                    m287 --(ownedBody)-> m288
1302            restrictions
1303                m288 derived from m284
1304                m287 derived from m284
1305        end rule
1306
1307        rule 'GeneratorEvalCreates Body if there is some operationSort'
```

```
1308        match with
1309            m289: any SATEL.APN.adtmm::Operation
1310            m290: existing SATEL.APN.adtmm::AtomicSort
1311            m291: any SATEL.APN.adtmm::ADT
1312            subject to
1313                m289 --(operationSorts)-> m290
1314                m291 --(ownedGenerators)-> m289
1315        apply
1316            m292:
1317                    mprologTermReference::Clause(
1318                        self = 'GeneratorClauseEval'
1319                    )
1320            m293:
1321                    mprologTermReference::Body(
1322                        self = 'GeneratorBodyEval'
1323                    )
1324            subject to
1325                m292 --(ownedBody)-> m293
1326        restrictions
1327            m293 derived from m289
1328            m292 derived from m289
1329    end rule
1330
1331    rule 'Generators inside Model'
1332        match with
1333            m294: any SATEL.APN.adtmm::Operation
1334            m295: any SATEL.APN.adtmm::ADT
1335            subject to
1336                m295 --(ownedGenerators)-> m294
1337        apply
1338            m296: mprologTermReference::Clause(
1339                        self = 'GeneratorClauseEval'
1340                    )
1341            m297: mprologTermReference::Head(
1342                        self = 'GeneratorHeadEval'
1343                    )
1344            subject to
1345                m296 --(ownedHead)-> m297
1346        restrictions
1347            m296 derived from m294
1348            m297 derived from m294
1349    end rule
1350
1351    rule 'Generators Eval inside Model'
1352        match with
1353            m298: any SATEL::Model
1354            m299: any SATEL.APN.adtmm::Operation
1355            subject to
1356                m298 ~~(contains)~> m299
1357        apply
1358            m300: mprologTermReference::Clause(
1359                        self = 'GeneratorClauseEval'
1360                    )
1361            m301: mprologTermReference::Model(
1362                        ApplyAttribute= 'Model'
1363                    )
1364            subject to
```

```
1365                m301 --(ownedClause)-> m300
1366            restrictions
1367                m300 derived from m299
1368                m301 derived from m298
1369        end rule
1370
1371        rule 'Generators BodyEvalFunctors'
1372            match with
1373                m302: any SATEL.APN.adtmm::Operation
1374                m303: any SATEL.APN.adtmm::AtomicSort
1375                m304: any SATEL.APN.adtmm::ADT
1376                subject to
1377                    m302 --(operationSorts)-> m303
1378                    m304 --(ownedGenerators)-> m302
1379
1380            apply
1381                m305: mprologTermReference::Body(
1382                            ApplyAttribute= 'GeneratorBodyEval'
1383                        )
1384                m306: mprologTermReference::Functor(
1385                            ApplyAttribute= 'SortEvalFunctor'
1386                        )
1387                subject to
1388                    m305 --(ownedTerm)-> m306
1389                    m306 --(nextTerm)-> m306
1390
1391            restrictions
1392                m305 derived from m302
1393                m306 derived from m303
1394                m306 derived from m302
1395        end rule
1396
1397        rule 'Generators BodyEvalFunctors'
1398            match with
1399                m307: any SATEL.APN.adtmm::Operation
1400                m308: any SATEL.APN.adtmm::AtomicSort
1401                m309: any SATEL.APN.adtmm::ADT
1402
1403                subject to
1404                    m307 --(operationSorts)-> m308
1405                    m309 --(ownedGenerators)-> m307
1406
1407            apply
1408                m310: mprologTermReference::Functor(
1409                            ApplyAttribute= 'GeneratorEvalFirst'
1410                        )
1411                m311: mprologTermReference::Variable(
1412                            ApplyAttribute= 'SortVar'
1413                        )
1414                subject to
1415                    m310 --(ownedTerm)-> m311
1416                    m311 --(nextTerm)-> m311
1417
1418            restrictions
1419                m310 derived from m307
1420                m311 derived from m308
1421                m311 derived from m307
```

```
1422    end rule
1423
1424    rule 'Generators BodyEvalFunctors'
1425        match with
1426            m312:any SATEL.APN.adtmm::Operation
1427            m313: any SATEL.APN.adtmm::AtomicSort
1428            m314: any SATEL.APN.adtmm::ADT
1429            subject to
1430                m312 --(operationSorts)-> m313
1431                m314 --(ownedGenerators)-> m312
1432        apply
1433            m315: mprologTermReference::Functor(
1434                        ApplyAttribute= 'GeneratorEvalSecond'
1435                    )
1436            m316: mprologTermReference::Variable(
1437                        ApplyAttribute= 'SortVarAfterEval'
1438                    )
1439            subject to
1440                m315 --(ownedTerm)-> m316
1441                m316 --(nextTerm)-> m316
1442        restrictions
1443            m315 derived from m312
1444            m316 derived from m313
1445            m316 derived from m312
1446    end rule
1447
1448    rule 'InequationRightTerm'
1449        match with
1450            m317: any SATEL.APN.adtmm::Inequation
1451            m318: any SATEL.APN.adtmm::Term
1452            subject to
1453                m317 --(ownedRightTerm)-> m318
1454        apply
1455            m319: mprologTermReference::Functor(
1456                        self = 'RightEvalFunctor'
1457                    )
1458            m320: mprologTermReference::Functor(
1459                        self = 'TermFunctor'
1460                    )
1461            m321: mprologTermReference::Variable(
1462                        self = 'REvalFunctorVariable'
1463                    )
1464            subject to
1465                m319 --(ownedTerm)-> m320
1466                m320 --(nextTerm)-> m321
1467        restrictions
1468            m319 derived from m317
1469            m320 derived from m318
1470            m321 derived from m317
1471    end rule
1472
1473    rule 'Inequation next AbstractEquation'
1474        match with
1475            m322: any SATEL.APN.adtmm::Inequation
1476            m323: any SATEL.APN.adtmm::AbstractEquation
1477
1478            subject to
```

```
1479                m322 --(next)-> m323
1480            apply
1481                m324: mprologTermReference::Functor(
1482                            self = 'LeftEvalFunctor'
1483                        )
1484                m325: mprologTermReference::InfixExpression(
1485                            self = 'IneqInfixExp'
1486                        )
1487                subject to
1488                    m325 --(nextTerm)-> m324
1489            restrictions
1490                m324 derived from m323
1491                m325 derived from m322
1492        end rule
1493
1494        rule 'Set Right Side OwnedEquation  if the last condition is an inequation'
1495            match with
1496                m326: any SATEL.APN.adtmm::CondEquation
1497                m327: any SATEL.APN.adtmm::Inequation
1498                m328: not SATEL.APN.adtmm::AbstractEquation
1499                m329: any SATEL.APN.adtmm::Equation
1500                subject to
1501                    m327 !-(next)-> m328
1502                    m326 --(ownedEquation)-> m329
1503                    m326 --(ownedConditions)-> m327
1504            apply
1505                m330: mprologTermReference::InfixExpression(
1506                            self = 'IneqInfixExp'
1507                        )
1508                m331: mprologTermReference::Functor(
1509                            self = 'RightEvalFunctor'
1510                        )
1511                subject to
1512                    m330 --(nextTerm)-> m331
1513            restrictions
1514                m330 derived from m327
1515                m331 derived from m329
1516        end rule
1517
1518        rule 'AxiomClause in Model'
1519            match with
1520                m332: any SATEL::Model
1521                m333: any SATEL::Axiom
1522                subject to
1523                    m332 ~~(contains)~> m333
1524            apply
1525                m334:mprologTermReference::Clause(
1526                            self = 'AxiomClause'
1527                        )
1528                m335:mprologTermReference::Model(
1529                            ApplyAttribute= 'Model'
1530                        )
1531                subject to
1532                    m335 --(ownedClause)-> m334
1533            restrictions
1534                m334 derived from m333
1535                m335 derived from m332
```

```
1536    end rule
1537
1538
1539    rule 'ConditionAtom next ConditionAtom'
1540        match with
1541            m336:any SATEL::ConditionAtom
1542            m337:any SATEL::ConditionAtom
1543            subject to
1544                m336 --(next)-> m337
1545
1546        apply
1547            m338: mprologTermReference::Functor(
1548                        self = 'CAtomFunctor'
1549                    )
1550            m339: mprologTermReference::Functor(
1551                        ApplyAttribute= 'CAtomFunctor'
1552                    )
1553            m340: mprologTermReference::Functor(
1554                        self = 'CondAtomEvalFunctor'
1555                    )
1556            m341: mprologTermReference::Functor(
1557                        self = 'CondAtomEvalFunctor'
1558                    )
1559            m342: mprologTermReference::Functor(
1560                        text= 'true'
1561                    )
1562            m343: mprologTermReference::Functor(
1563                        text= 'true'
1564                    )
1565            subject to
1566                m341 --(nextTerm)-> m340
1567                m340 --(ownedTerm)-> m338
1568                m341 --(ownedTerm)-> m339
1569                m338 --(nextTerm)-> m343
1570                m339 --(nextTerm)-> m342
1571
1572        restrictions
1573            m338 derived from m337
1574            m339 derived from m336
1575            m341 derived from m336
1576            m340 derived from m337
1577    end rule
1578
1579
1580    rule 'Connect Functors of ConditionAtom'
1581        match with
1582            m344:
1583                    any SATEL::ConditionBody
1584            m345:
1585                    any SATEL::ConditionAtom
1586            m346:
1587                    not SATEL::ConditionAtom
1588
1589            subject to
1590                m344 --(conditionAtom)-> m345
1591                m346 !-(next)-> m345
1592
```

```
1593
1594        apply
1595            m347:
1596                    mprologTermReference::Functor(
1597                        self = 'CondAtomEvalFunctor'
1598                    )
1599            m348:
1600                    mprologTermReference::Body(
1601                        ApplyAttribute= 'CondBodyFunctor'
1602                    )
1603            subject to
1604                m348 --(ownedTerm)-> m347
1605
1606        restrictions
1607            m347 derived from m345
1608            m348 derived from m344
1609            m347 derived from m344
1610
1611
1612
1613    end rule
1614
1615
1616    rule 'AxiomBody in Clause'
1617        match with
1618            m349:
1619                    any SATEL::ConditionBody
1620            m350:
1621                    any SATEL::Axiom
1622
1623            subject to
1624                m350 ~~(contains)~> m349
1625
1626
1627        apply
1628            m351:
1629                    mprologTermReference::Clause(
1630                        self = 'AxiomClause'
1631                    )
1632            m352:
1633                    mprologTermReference::Body(
1634                        ApplyAttribute= 'CondBodyFunctor'
1635                    )
1636            subject to
1637                m351 --(ownedBody)-> m352
1638
1639        restrictions
1640            m351 derived from m350
1641            m352 derived from m349
1642
1643
1644
1645    end rule
1646
1647
1648    rule 'CompositeTerm first ownedTerm'
1649        match with
```

```
            m353:
                    any SATEL.AlgebraicExpressions.algterms::CompositeTerm
            m354:
                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
            m355:
                    not SATEL.AlgebraicExpressions.algterms::AlgebraicTerm

            subject to
                m355 !-(next)-> m354
                m353 --(terms)-> m354


        apply
            m356:
                    mprologTermReference::Functor(
                        self = 'AlgTermFunctor'
                    )
            m357:
                    mprologTermReference::Functor(
                        self = 'AlgTermFunctor'
                    )
            subject to
                m357 --(ownedTerm)-> m356

        restrictions
            m356 derived from m354
            m357 derived from m353



    end rule


    rule 'AlgebraicTerm next AlgebraicTerm'
        match with
            m358:
                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
            m359:
                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm

            subject to
                m358 --(next)-> m359


        apply
            m360:
                    mprologTermReference::Functor(
                        self = 'AlgTermFunctor'
                    )
            m361:
                    mprologTermReference::Functor(
                        self = 'AlgTermFunctor'
                    )
            subject to
                m361 --(nextTerm)-> m360

        restrictions
```

```
1707            m360 derived from m359
1708            m361 derived from m358
1709
1710
1711
1712    end rule
1713
1714
1715    rule 'Parameter next Parameter'
1716        match with
1717            m362:
1718                    any SATEL.HMLFormula::Parameter
1719            m363:
1720                    any SATEL.HMLFormula::Parameter
1721            m364:
1722                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
1723            m365:
1724                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
1725
1726            subject to
1727                m362 --(next)-> m363
1728                m363 --(value)-> m364
1729                m362 --(value)-> m365
1730
1731
1732        apply
1733            m366:
1734                    mprologTermReference::Functor(
1735                        self = 'AlgTermFunctor'
1736                    )
1737            m367:
1738                    mprologTermReference::Functor(
1739                        self = 'AlgTermFunctor'
1740                    )
1741            subject to
1742                m367 --(nextTerm)-> m366
1743
1744        restrictions
1745            m366 derived from m364
1746            m367 derived from m365
1747
1748
1749
1750    end rule
1751
1752
1753    rule 'AlgEquality Args'
1754        match with
1755            m368:
1756                    any SATEL.AlgebraicExpressions::AlgEquality
1757            m369:
1758                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
1759            m370:
1760                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
1761
1762            subject to
1763                m368 --(algebraicTermL)-> m369
```

```
1764                m368 --(algebraicTermR)-> m370
1765
1766
1767            apply
1768                m371:
1769                        mprologTermReference::Functor(
1770                            self = 'AlgTermFunctor'
1771                        )
1772                m372:
1773                        mprologTermReference::Functor(
1774                            self = 'AlgTermFunctor'
1775                        )
1776                m373:
1777                        mprologTermReference::Functor(
1778                            self = 'CAtomFunctor'
1779                        )
1780                subject to
1781                    m371 --(nextTerm)-> m372
1782                    m373 --(ownedTerm)-> m371
1783
1784            restrictions
1785                m371 derived from m369
1786                m372 derived from m370
1787                m373 derived from m368
1788
1789
1790
1791    end rule
1792
1793
1794    rule 'SynchronizationEquality'
1795        match with
1796            m374:
1797                    any SATEL.AlgebraicExpressions::SyncEquality
1798            m375:
1799                    any SATEL.HMLFormula::SynchronizationTerm
1800            m376:
1801                    any SATEL.HMLFormula::SynchronizationTerm
1802
1803            subject to
1804                m374 --(synchronizationTermL)-> m375
1805                m374 --(synchronizationTermR)-> m376
1806
1807
1808        apply
1809            m377:
1810                    mprologTermReference::Term(
1811                        self = 'syncTerm'
1812                    )
1813            m378:
1814                    mprologTermReference::Term(
1815                        self = 'syncTerm'
1816                    )
1817            m379:
1818                    mprologTermReference::Functor(
1819                        self = 'CAtomFunctor'
1820                    )
```

```
1821            subject to
1822                m377 --(nextTerm)-> m378
1823                m379 --(ownedTerm)-> m377
1824
1825        restrictions
1826            m377 derived from m375
1827            m378 derived from m376
1828            m379 derived from m374
1829
1830
1831
1832    end rule
1833
1834
1835    rule 'HMLEquality'
1836        match with
1837            m380:
1838                    any SATEL.AlgebraicExpressions::HMLEquality
1839            m381:
1840                    any SATEL.HMLFormula::HMLTerm
1841            m382:
1842                    any SATEL.HMLFormula::HMLTerm
1843
1844            subject to
1845                m380 --(hmlTernL)-> m381
1846                m380 --(hmlTermR)-> m382
1847
1848
1849        apply
1850            m383:
1851                    mprologTermReference::Term(
1852                        self = 'hmlTerm'
1853                    )
1854            m384:
1855                    mprologTermReference::Term(
1856                        self = 'hmlTerm'
1857                    )
1858            m385:
1859                    mprologTermReference::Functor(
1860                        self = 'CAtomFunctor'
1861                    )
1862            subject to
1863                m383 --(nextTerm)-> m384
1864                m385 --(ownedTerm)-> m383
1865
1866        restrictions
1867            m383 derived from m381
1868            m384 derived from m382
1869            m385 derived from m380
1870
1871
1872
1873    end rule
1874
1875
1876    rule 'ArithmeticTerm'
1877        match with
```

```
1878            m386:
1879                    any SATEL.AlgebraicExpressions::ArithmeticEquality
1880            m387:
1881                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
1882            m388:
1883                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
1884
1885            subject to
1886                m386 --(arithmeticTermL)-> m387
1887                m386 --(arithmeticTermR)-> m388
1888
1889
1890        apply
1891            m389:
1892                    mprologTermReference::Term(
1893                        self = 'ArithmeticFunctor'
1894                    )
1895            m390:
1896                    mprologTermReference::Term(
1897                        self = 'ArithmeticFunctor'
1898                    )
1899            m391:
1900                    mprologTermReference::Functor(
1901                        self = 'CAtomFunctor'
1902                    )
1903            subject to
1904                m389 --(nextTerm)-> m390
1905                m391 --(ownedTerm)-> m389
1906
1907        restrictions
1908            m389 derived from m387
1909            m390 derived from m388
1910            m391 derived from m386
1911
1912
1913
1914    end rule
1915
1916
1917    rule 'BooleanEquality'
1918        match with
1919            m392:
1920                    any SATEL.AlgebraicExpressions::BooleanEquality
1921            m393:
1922                    any SATEL.AlgebraicExpressions.booleanterms::BooleanTerm
1923            m394:
1924                    any SATEL.AlgebraicExpressions.booleanterms::BooleanTerm
1925
1926            subject to
1927                m392 --(booleanTermL)-> m393
1928                m392 --(booleanTermR)-> m394
1929
1930
1931        apply
1932            m395:
1933                    mprologTermReference::Term(
1934                        self = 'CAtomFunctor'
```

```
1935                    )
1936            m396:
1937                    mprologTermReference::Term(
1938                        self = 'CAtomFunctor'
1939                    )
1940            m397:
1941                    mprologTermReference::Functor(
1942                        self = 'CAtomFunctor'
1943                    )
1944            subject to
1945                m395 --(nextTerm)-> m396
1946                m397 --(ownedTerm)-> m395
1947
1948        restrictions
1949            m395 derived from m393
1950            m396 derived from m394
1951            m397 derived from m392
1952
1953
1954
1955    end rule
1956
1957
1958    rule 'Not with BooleanTerm'
1959        match with
1960            m398:
1961                    any SATEL.AlgebraicExpressions.booleanterms::Not
1962            m399:
1963                    any SATEL.AlgebraicExpressions.booleanterms::BooleanTerm
1964
1965            subject to
1966                m398 --(booleanTerm)-> m399
1967
1968
1969        apply
1970            m400:
1971                    mprologTermReference::Term(
1972                        self = 'CAtomFunctor'
1973                    )
1974            m401:
1975                    mprologTermReference::Functor(
1976                        self = 'CAtomFunctor'
1977                    )
1978            subject to
1979                m401 --(ownedTerm)-> m400
1980
1981        restrictions
1982            m400 derived from m399
1983            m401 derived from m398
1984
1985
1986
1987    end rule
1988
1989
1990    rule 'Sequence with hmlTerm'
1991        match with
```

```
            m402:
                    any SATEL.AlgebraicExpressions.booleanterms::Sequence
            m403:
                    any SATEL.HMLFormula::HMLTerm

            subject to
                m402 --(hmlTerm)-> m403


        apply
            m404:
                    mprologTermReference::Term(
                        self = 'hmlTerm'
                    )
            m405:
                    mprologTermReference::Functor(
                        self = 'CAtomFunctor'
                    )
            subject to
                m405 --(ownedTerm)-> m404

        restrictions
            m404 derived from m403
            m405 derived from m402



    end rule


    rule 'Positive with HMLTerm'
        match with
            m406:
                    any SATEL.AlgebraicExpressions.booleanterms::Positive
            m407:
                    any SATEL.HMLFormula::HMLTerm

            subject to
                m406 --(hmlTerm)-> m407


        apply
            m408:
                    mprologTermReference::Term(
                        self = 'hmlTerm'
                    )
            m409:
                    mprologTermReference::Functor(
                        self = 'CAtomFunctor'
                    )
            subject to
                m409 --(ownedTerm)-> m408

        restrictions
            m408 derived from m407
            m409 derived from m406
```

```
2049
2050
2051    end rule
2052
2053
2054    rule 'Trace with HMLTerm'
2055        match with
2056            m410:
2057                    any SATEL.AlgebraicExpressions.booleanterms::Trace
2058            m411:
2059                    any SATEL.HMLFormula::HMLTerm
2060
2061            subject to
2062                m410 --(hmlTerm)-> m411
2063
2064
2065        apply
2066            m412:
2067                    mprologTermReference::Term(
2068                        self = 'hmlTerm'
2069                    )
2070            m413:
2071                    mprologTermReference::Functor(
2072                        self = 'CAtomFunctor'
2073                    )
2074            subject to
2075                m413 --(ownedTerm)-> m412
2076
2077        restrictions
2078            m412 derived from m411
2079            m413 derived from m410
2080
2081
2082
2083    end rule
2084
2085
2086    rule 'BOPAnd with BooleanTerms'
2087        match with
2088            m414:
2089                    any SATEL.AlgebraicExpressions.booleanterms::BOPAnd
2090            m415:
2091                    any SATEL.AlgebraicExpressions.booleanterms::BooleanTerm
2092            m416:
2093                    any SATEL.AlgebraicExpressions.booleanterms::BooleanTerm
2094
2095            subject to
2096                m414 --(booleanTermL)-> m415
2097                m414 --(booleanTermR)-> m416
2098
2099
2100        apply
2101            m417:
2102                    mprologTermReference::Term(
2103                        self = 'CAtomFunctor'
2104                    )
2105            m418:
```

```
2106                    mprologTermReference::Functor(
2107                        self = 'CAtomFunctor'
2108                    )
2109            m419:
2110                    mprologTermReference::Term(
2111                        self = 'CAtomFunctor'
2112                    )
2113            subject to
2114                m418 --(ownedTerm)-> m417
2115                m417 --(nextTerm)-> m419
2116
2117        restrictions
2118            m417 derived from m415
2119            m418 derived from m414
2120            m419 derived from m416
2121
2122
2123
2124    end rule
2125
2126
2127    rule 'BOPOr with BooleanTerms'
2128        match with
2129            m420:
2130                    any SATEL.AlgebraicExpressions.booleanterms::BOPOr
2131            m421:
2132                    any SATEL.AlgebraicExpressions.booleanterms::BooleanTerm
2133            m422:
2134                    any SATEL.AlgebraicExpressions.booleanterms::BooleanTerm
2135
2136            subject to
2137                m420 --(booleanTermL)-> m421
2138                m420 --(booleanTermR)-> m422
2139
2140
2141        apply
2142            m423:
2143                    mprologTermReference::Term(
2144                        self = 'CAtomFunctor'
2145                    )
2146            m424:
2147                    mprologTermReference::Functor(
2148                        self = 'CAtomFunctor'
2149                    )
2150            m425:
2151                    mprologTermReference::Term(
2152                        self = 'CAtomFunctor'
2153                    )
2154            subject to
2155                m424 --(ownedTerm)-> m423
2156                m423 --(nextTerm)-> m425
2157
2158        restrictions
2159            m423 derived from m421
2160            m424 derived from m420
2161            m425 derived from m422
2162
```

```
2163
2164
2165    end rule
2166
2167
2168    rule 'Equals with BooleanTerms'
2169        match with
2170            m426:
2171                    any SATEL.AlgebraicExpressions.booleanterms::Equals
2172            m427:
2173                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2174            m428:
2175                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2176
2177            subject to
2178                m426 --(arithmeticTermL)-> m427
2179                m426 --(arithmeticTermR)-> m428
2180
2181
2182        apply
2183            m429:
2184                    mprologTermReference::Term(
2185                        self = 'ArithmeticFunctor'
2186                    )
2187            m430:
2188                    mprologTermReference::Functor(
2189                        self = 'CAtomFunctor'
2190                    )
2191            m431:
2192                    mprologTermReference::Term(
2193                        self = 'ArithmeticFunctor'
2194                    )
2195            subject to
2196                m430 --(ownedTerm)-> m429
2197                m429 --(nextTerm)-> m431
2198
2199        restrictions
2200            m429 derived from m427
2201            m430 derived from m426
2202            m431 derived from m428
2203
2204
2205
2206    end rule
2207
2208
2209    rule 'NotEquals with ArithmeticTerms'
2210        match with
2211            m432:
2212                    any SATEL.AlgebraicExpressions.booleanterms::NotEquals
2213            m433:
2214                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2215            m434:
2216                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2217
2218            subject to
2219                m432 --(arithmeticTermL)-> m433
```

```
2220                m432 --(arithmeticTermR)-> m434
2221
2222
2223            apply
2224                m435:
2225                        mprologTermReference::Term(
2226                            self = 'ArithmeticFunctor'
2227                        )
2228                m436:
2229                        mprologTermReference::Functor(
2230                            self = 'CAtomFunctor'
2231                        )
2232                m437:
2233                        mprologTermReference::Term(
2234                            self = 'ArithmeticFunctor'
2235                        )
2236                subject to
2237                    m436 --(ownedTerm)-> m435
2238                    m435 --(nextTerm)-> m437
2239
2240            restrictions
2241                m435 derived from m433
2242                m436 derived from m432
2243                m437 derived from m434
2244
2245
2246
2247        end rule
2248
2249
2250        rule 'LT with ArithmeticTerms'
2251            match with
2252                m438:
2253                        any SATEL.AlgebraicExpressions.booleanterms::LT
2254                m439:
2255                        any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2256                m440:
2257                        any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2258
2259                subject to
2260                    m438 --(arithmeticTermL)-> m439
2261                    m438 --(arithmeticTermR)-> m440
2262
2263
2264            apply
2265                m441:
2266                        mprologTermReference::Term(
2267                            self = 'ArithmeticFunctor'
2268                        )
2269                m442:
2270                        mprologTermReference::Functor(
2271                            self = 'CAtomFunctor'
2272                        )
2273                m443:
2274                        mprologTermReference::Term(
2275                            self = 'ArithmeticFunctor'
2276                        )
```

```
2277            subject to
2278                m442 --(ownedTerm)-> m441
2279                m441 --(nextTerm)-> m443
2280
2281        restrictions
2282            m441 derived from m439
2283            m442 derived from m438
2284            m443 derived from m440
2285
2286
2287
2288    end rule
2289
2290
2291    rule 'GT with ArithmeticTerm'
2292        match with
2293            m444:
2294                    any SATEL.AlgebraicExpressions.booleanterms::GT
2295            m445:
2296                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2297            m446:
2298                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2299
2300            subject to
2301                m444 --(arithmeticTermL)-> m445
2302                m444 --(arithmeticTermR)-> m446
2303
2304
2305        apply
2306            m447:
2307                    mprologTermReference::Term(
2308                        self = 'ArithmeticFunctor'
2309                    )
2310            m448:
2311                    mprologTermReference::Functor(
2312                        self = 'CAtomFunctor'
2313                    )
2314            m449:
2315                    mprologTermReference::Term(
2316                        self = 'ArithmeticFunctor'
2317                    )
2318            subject to
2319                m448 --(ownedTerm)-> m447
2320                m447 --(nextTerm)-> m449
2321
2322        restrictions
2323            m447 derived from m445
2324            m448 derived from m444
2325            m449 derived from m446
2326
2327
2328
2329    end rule
2330
2331
2332    rule 'GTE with ArithmeticTerm'
2333        match with
```

```
2334            m450:
2335                    any SATEL.AlgebraicExpressions.booleanterms::GTE
2336            m451:
2337                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2338            m452:
2339                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2340
2341            subject to
2342                m450 --(arithmeticTermL)-> m451
2343                m450 --(arithmeticTermR)-> m452
2344
2345
2346        apply
2347            m453:
2348                    mprologTermReference::Term(
2349                        self = 'ArithmeticFunctor'
2350                    )
2351            m454:
2352                    mprologTermReference::Functor(
2353                        self = 'CAtomFunctor'
2354                    )
2355            m455:
2356                    mprologTermReference::Term(
2357                        self = 'ArithmeticFunctor'
2358                    )
2359            subject to
2360                m454 --(ownedTerm)-> m453
2361                m453 --(nextTerm)-> m455
2362
2363        restrictions
2364            m453 derived from m451
2365            m454 derived from m450
2366            m455 derived from m452
2367
2368
2369
2370    end rule
2371
2372
2373    rule 'LTE with ArithmeticTerm'
2374        match with
2375            m456:
2376                    any SATEL.AlgebraicExpressions.booleanterms::LTE
2377            m457:
2378                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2379            m458:
2380                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2381
2382            subject to
2383                m456 --(arithmeticTermL)-> m457
2384                m456 --(arithmeticTermR)-> m458
2385
2386
2387        apply
2388            m459:
2389                    mprologTermReference::Term(
2390                        self = 'ArithmeticFunctor'
```

```
                    )
            m460:
                    mprologTermReference::Functor(
                        self = 'CAtomFunctor'
                    )
            m461:
                    mprologTermReference::Term(
                        self = 'ArithmeticFunctor'
                    )
            subject to
                m460 --(ownedTerm)-> m459
                m459 --(nextTerm)-> m461

        restrictions
            m459 derived from m457
            m460 derived from m456
            m461 derived from m458



    end rule


    rule 'BOPPlus with ArithmeticTerm'
        match with
            m462:
                    any SATEL.AlgebraicExpressions.arithmeticterms::BOPPlus
            m463:
                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
            m464:
                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm

            subject to
                m462 --(arithmeticTermL)-> m463
                m462 --(arithmeticTermR)-> m464


        apply
            m465:
                    mprologTermReference::Term(
                        self = 'ArithmeticFunctor'
                    )
            m466:
                    mprologTermReference::Functor(
                        self = 'ArithmeticTerm'
                    )
            m467:
                    mprologTermReference::Term(
                        self = 'ArithmeticFunctor'
                    )
            subject to
                m466 --(ownedTerm)-> m465
                m465 --(nextTerm)-> m467

        restrictions
            m465 derived from m463
            m466 derived from m462
```

```
2448            m467 derived from m464
2449
2450
2451
2452    end rule
2453
2454
2455    rule 'BOPMinus with ArithmeticTerm'
2456        match with
2457            m468:
2458                    any SATEL.AlgebraicExpressions.arithmeticterms::BOPMinus
2459            m469:
2460                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2461            m470:
2462                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2463
2464            subject to
2465                m468 --(arithmeticTermL)-> m469
2466                m468 --(arithmeticTermR)-> m470
2467
2468
2469        apply
2470            m471:
2471                    mprologTermReference::Term(
2472                        self = 'ArithmeticFunctor'
2473                    )
2474            m472:
2475                    mprologTermReference::Functor(
2476                        self = 'ArithmeticTerm'
2477                    )
2478            m473:
2479                    mprologTermReference::Term(
2480                        self = 'ArithmeticFunctor'
2481                    )
2482            subject to
2483                m472 --(ownedTerm)-> m471
2484                m471 --(nextTerm)-> m473
2485
2486        restrictions
2487            m471 derived from m469
2488            m472 derived from m468
2489            m473 derived from m470
2490
2491
2492
2493    end rule
2494
2495
2496    rule 'BOPTimes with ArithmeticTerm'
2497        match with
2498            m474:
2499                    any SATEL.AlgebraicExpressions.arithmeticterms::BOPTimes
2500            m475:
2501                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2502            m476:
2503                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2504
```

```
2505            subject to
2506                m474 --(arithmeticTermL)-> m475
2507                m474 --(arithmeticTermR)-> m476
2508
2509
2510        apply
2511            m477:
2512                    mprologTermReference::Term(
2513                        self = 'ArithmeticFunctor'
2514                    )
2515            m478:
2516                    mprologTermReference::Functor(
2517                        self = 'ArithmeticFunctor'
2518                    )
2519            m479:
2520                    mprologTermReference::Term(
2521                        self = 'ArithmeticFunctor'
2522                    )
2523            subject to
2524                m478 --(ownedTerm)-> m477
2525                m477 --(nextTerm)-> m479
2526
2527        restrictions
2528            m477 derived from m475
2529            m478 derived from m474
2530            m479 derived from m476
2531
2532
2533
2534    end rule
2535
2536
2537    rule 'BOPDiv with ArithmeticTerm'
2538        match with
2539            m480:
2540                    any SATEL.AlgebraicExpressions.arithmeticterms::BOPDiv
2541            m481:
2542                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2543            m482:
2544                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2545
2546            subject to
2547                m480 --(arithmeticTermL)-> m481
2548                m480 --(arithmeticTermR)-> m482
2549
2550
2551        apply
2552            m483:
2553                    mprologTermReference::Term(
2554                        self = 'ArithmeticFunctor'
2555                    )
2556            m484:
2557                    mprologTermReference::Functor(
2558                        self = 'ArithmeticFunctor'
2559                    )
2560            m485:
2561                    mprologTermReference::Term(
```

```
                    self = 'ArithmeticFunctor'
                )
        subject to
            m484 --(ownedTerm)-> m483
            m483 --(nextTerm)-> m485

    restrictions
        m483 derived from m481
        m484 derived from m480
        m485 derived from m482



end rule


rule 'NBEvents with HMLTerm'
    match with
        m486:
                any SATEL.AlgebraicExpressions.arithmeticterms::NBEvents
        m487:
                any SATEL.HMLFormula::HMLTerm

        subject to
            m486 --(hmlTerm)-> m487


    apply
        m488:
                mprologTermReference::List(
                    self = 'hmlTermList'
                )
        m489:
                mprologTermReference::Functor(
                    self = 'ArithmeticFunctor'
                )
        subject to
            m489 --(ownedTerm)-> m488

    restrictions
        m488 derived from m487
        m489 derived from m486



end rule


rule 'Depth with HMLTerm'
    match with
        m490:
                any SATEL.AlgebraicExpressions.arithmeticterms::Depth
        m491:
                any SATEL.HMLFormula::HMLTerm

        subject to
            m490 --(hmlTerm)-> m491
```

```
2619
2620
2621        apply
2622            m492:
2623                    mprologTermReference::List(
2624                        self = 'hmlTermList'
2625                    )
2626            m493:
2627                    mprologTermReference::Functor(
2628                        self = 'ArithmeticFunctor'
2629                    )
2630            subject to
2631                m493 --(ownedTerm)-> m492
2632
2633        restrictions
2634            m492 derived from m491
2635            m493 derived from m490
2636
2637
2638
2639    end rule
2640
2641
2642    rule 'UOPMinus with ArithmeticTerm'
2643        match with
2644            m494:
2645                    any SATEL.AlgebraicExpressions.arithmeticterms::UOPMinus
2646            m495:
2647                    any SATEL.AlgebraicExpressions.arithmeticterms::ArithmeticTerm
2648
2649            subject to
2650                m494 --(arithmeticTerm)-> m495
2651
2652
2653        apply
2654            m496:
2655                    mprologTermReference::Term(
2656                        self = 'ArithmeticFunctor'
2657                    )
2658            m497:
2659                    mprologTermReference::Functor(
2660                        self = 'ArithmeticFunctor'
2661                    )
2662            subject to
2663                m497 --(ownedTerm)-> m496
2664
2665        restrictions
2666            m496 derived from m495
2667            m497 derived from m494
2668
2669
2670
2671    end rule
2672
2673
2674    rule 'InclusionWithHMLTermAndIntentionDec'
2675        match with
```

```
            m498:
                    any SATEL::Inclusion
            m499:
                    any SATEL::IntentionDec
            m500:
                    any SATEL.HMLFormula::HMLTerm

            subject to
                m498 --(_in)-> m499
                m498 --(hmlTerm)-> m500


        apply
            m501:
                    mprologTermReference::QuotedAtom(
                        self = 'intDecQA'
                    )
            m502:
                    mprologTermReference::Functor(
                        self = 'CAtomFunctor'
                    )
            m503:
                    mprologTermReference::List(
                        self = 'hmlTermList'
                    )
            subject to
                m502 --(ownedTerm)-> m503
                m503 --(nextTerm)-> m501

        restrictions
            m501 derived from m499
            m502 derived from m498
            m503 derived from m500
            m501 derived from m498



    end rule


    rule 'AxiomClause in Model'
        match with
            m504:
                    any SATEL::Axiom
            m505:
                    any SATEL::Inclusion

            subject to
                m504 --(inclusion)-> m505


        apply
            m506:
                    mprologTermReference::Head(
                        self = 'AxiomClauseHead'
                    )
            m507:
```

```
2733                    mprologTermReference::Functor(
2734                        ApplyAttribute= 'CAtomFunctor'
2735                    )
2736            subject to
2737                m506 --(ownedFunctor)-> m507
2738
2739        restrictions
2740            m506 derived from m504
2741            m507 derived from m505
2742
2743
2744
2745    end rule
2746
2747
2748    rule 'HMLTermWithFirstHMLFormula'
2749        match with
2750            m508:
2751                    any SATEL.HMLFormula::HMLTerm
2752            m509:
2753                    any SATEL.HMLFormula::HMLFormula
2754            m510:
2755                    not SATEL.HMLFormula::HMLFormula
2756
2757            subject to
2758                m508 --(hmlFormula)-> m509
2759                m510 !-(next)-> m509
2760
2761
2762        apply
2763            m511:
2764                    mprologTermReference::Term(
2765                        self = 'hmlFormulaFunctor'
2766                    )
2767            m512:
2768                    mprologTermReference::List(
2769                        self = 'hmlTermList'
2770                    )
2771            subject to
2772                m512 --(ownedHeadTerms)-> m511
2773
2774        restrictions
2775            m511 derived from m509
2776            m512 derived from m508
2777
2778
2779
2780    end rule
2781
2782
2783    rule 'HMLNext with Event and nextFormula'
2784        match with
2785            m513:
2786                    any SATEL.HMLFormula::HMLNext
2787            m514:
2788                    any SATEL.HMLFormula::HMLEvent
2789            m515:
```

```
                    any SATEL.HMLFormula::HMLFormulaContent

            subject to
                m513 --(hmlEvent)-> m514
                m513 --(hmlFormulaContent)-> m515


        apply
            m516:
                    mprologTermReference::Functor(
                        self = 'eventFunctor'
                    )
            m517:
                    mprologTermReference::Functor(
                        self = 'hmlFormulaFunctor'
                    )
            m518:
                    mprologTermReference::Functor(
                        self = 'hmlFormulaFunctor'
                    )
            subject to
                m517 --(ownedTerm)-> m516
                m516 --(nextTerm)-> m518

        restrictions
            m516 derived from m514
            m517 derived from m513
            m518 derived from m515



    end rule


    rule 'HMLEventWithInputOnly'
        match with
            m519:
                    any SATEL.HMLFormula::HMLEvent
            m520:
                    any SATEL.HMLFormula::SynchronizationInputTerm
            m521:
                    not SATEL.HMLFormula::SynchronizationOutputTerm

            subject to
                m519 --(inputTerm)-> m520
                m519 !-(outputTerm)-> m521


        apply
            m522:
                    mprologTermReference::Term(
                        self = 'syncTerm'
                    )
            m523:
                    mprologTermReference::Functor(
                        self = 'eventFunctor'
                    )
```

```
2847            subject to
2848                m523 --(ownedTerm)-> m522
2849
2850        restrictions
2851            m522 derived from m520
2852            m523 derived from m519
2853
2854
2855
2856    end rule
2857
2858
2859    rule 'HMLEventWithInputAndOutput'
2860        match with
2861            m524:
2862                    any SATEL.HMLFormula::HMLEvent
2863            m525:
2864                    any SATEL.HMLFormula::SynchronizationInputTerm
2865            m526:
2866                    any SATEL.HMLFormula::SynchronizationOutputTerm
2867
2868            subject to
2869                m524 --(inputTerm)-> m525
2870                m524 --(outputTerm)-> m526
2871
2872
2873        apply
2874            m527:
2875                    mprologTermReference::Term(
2876                        self = 'syncTerm'
2877                    )
2878            m528:
2879                    mprologTermReference::Functor(
2880                        self = 'eventFunctor'
2881                    )
2882            m529:
2883                    mprologTermReference::Term(
2884                        self = 'syncTerm'
2885                    )
2886            subject to
2887                m528 --(ownedTerm)-> m527
2888                m527 --(nextTerm)-> m529
2889
2890        restrictions
2891            m527 derived from m525
2892            m528 derived from m524
2893            m529 derived from m526
2894
2895
2896
2897    end rule
2898
2899
2900    rule 'HMLFormula next HMLFormula'
2901        match with
2902            m530:
2903                    any SATEL.HMLFormula::HMLFormula
```

```
2904            m531:
2905                    any SATEL.HMLFormula::HMLFormula
2906
2907            subject to
2908                m530 --(next)-> m531
2909
2910
2911        apply
2912            m532:
2913                    mprologTermReference::Term(
2914                        self = 'hmlFormulaFunctor'
2915                    )
2916            m533:
2917                    mprologTermReference::Term(
2918                        self = 'hmlFormulaFunctor'
2919                    )
2920            subject to
2921                m533 --(nextTerm)-> m532
2922
2923        restrictions
2924            m532 derived from m531
2925            m533 derived from m530
2926
2927
2928
2929    end rule
2930
2931
2932    rule 'SynchronizationParameter'
2933        match with
2934            m534:
2935                    any SATEL.HMLFormula::SynchronizationEventOutputTerm
2936            m535:
2937                    any SATEL.HMLFormula::Parameter
2938            m536:
2939                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
2940
2941            subject to
2942                m534 --(parameters)-> m535
2943                m535 --(value)-> m536
2944
2945
2946        apply
2947            m537:
2948                    mprologTermReference::Term(
2949                        self = 'AlgTermFunctor'
2950                    )
2951            m538:
2952                    mprologTermReference::Functor(
2953                        self = 'syncTerm'
2954                    )
2955            subject to
2956                m538 --(ownedTerm)-> m537
2957
2958        restrictions
2959            m537 derived from m536
2960            m538 derived from m534
```

```
2961
2962
2963
2964    end rule
2965
2966
2967    rule 'Parameter next Parameter'
2968        match with
2969            m539:
2970                    any SATEL.HMLFormula::Parameter
2971            m540:
2972                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
2973            m541:
2974                    any SATEL.HMLFormula::Parameter
2975            m542:
2976                    any SATEL.AlgebraicExpressions.algterms::AlgebraicTerm
2977
2978            subject to
2979                m539 --(value)-> m540
2980                m541 --(value)-> m542
2981                m539 --(next)-> m541
2982
2983
2984        apply
2985            m543:
2986                    mprologTermReference::Term(
2987                        self = 'AlgTermFunctor'
2988                    )
2989            m544:
2990                    mprologTermReference::Term(
2991                        self = 'AlgTermFunctor'
2992                    )
2993            subject to
2994                m544 --(nextTerm)-> m543
2995
2996        restrictions
2997            m544 derived from m540
2998            m543 derived from m542
2999
3000    end rule
3001
3002
3003    rule 'MultisetFirstNumOfTerms_Term'
3004        match with
3005            ms:         any SATEL.APN.multisetmm::Multiset
3006            nt1:        any SATEL.APN.multisetmm::NumOfTerms
3007            nt2:        not SATEL.APN.multisetmm::NumOfTerms
3008            termadt: any SATEL.APN.adtmm::Term
3009
3010            subject to
3011                ms --(ownedNumOfTerms)-> nt1    // positive direct association
3012                nt2 !-(next)-> nt1                                  // negative match
                        association
3013                nt1 ~~(contains)~> termadt              //  positive indirect  association
3014
3015        apply
3016            term: mprologTermReference::Term(
```

```
3017                        self = 'TermFunctor'                    //ApplyAttribute
3018                        )
3019            list: mprologTermReference::List(
3020                        self = 'MSetList'
3021                        )
3022
3023            subject to
3024                list --(ownedHeadTerms)-> term
3025
3026        restrictions
3027            list derived from ms
3028            term derived from termadt
3029
3030    end rule
3031
3032
3033    rule 'InitializeInitialMarcationWithPlacesFunctors'
3034        match with
3035            m551:
3036                    any SATEL.APN.apnmm::APN
3037            m552:
3038                    any SATEL.APN.apnmm::Place
3039
3040            subject to
3041                m551 --(ownedPlaces)-> m552
3042
3043
3044        apply
3045            m553:
3046                    mprologTermReference::List(
3047                        self = 'InitialMarcationList'
3048                    )
3049            m554:
3050                    mprologTermReference::Functor(
3051                        self = 'PlaceFunctor'
3052                    )
3053            subject to
3054                m553 --(ownedHeadTerms)-> m554
3055                m554 --(nextTerm)-> m554
3056
3057        restrictions
3058            m553 derived from m551
3059            m554 derived from m552
3060    end rule
3061
3062    rule 'PlaceMultiset'
3063        match with
3064            m555:
3065                    any SATEL.APN.apnmm::Place
3066            m556:
3067                    any SATEL.APN.multisetmm::Multiset
3068
3069            subject to
3070                m555 --(ownedPlaceMultiset)-> m556
3071
3072
3073        apply
```

```
3074            m557:
3075                    mprologTermReference::Term(
3076                        self = 'PlaceOwnedTerm'
3077                    )
3078            m558:
3079                    mprologTermReference::List(
3080                        self = 'MSetList'
3081                    )
3082            subject to
3083                m557 --(nextTerm)-> m558
3084
3085        restrictions
3086            m557 derived from m555
3087            m558 derived from m556
3088
3089
3090
3091    end rule
3092
3093
3094    rule 'For  APN creates a Clause with Head'
3095        match with
3096            m559: any SATEL.APN.apnmm::APN
3097            m560:    any SATEL::Model
3098
3099            subject to
3100                m560 ~~(contains)~> m559
3101
3102        apply
3103            m561: mprologTermReference::Functor(
3104                        self = 'InitialMarcationFunctor'
3105                        )
3106            m562:  mprologTermReference::Model(
3107                        self = 'Model'
3108                    )
3109            m563: mprologTermReference::Clause
3110            m564: mprologTermReference::Head
3111
3112            subject to
3113                m562 --(ownedClause)-> m563
3114                m563 --(ownedHead)-> m564
3115                m564 --(ownedFunctor)-> m561
3116
3117        restrictions
3118            m561 derived from m559
3119            m562 derived from m560
3120
3121    end rule
3122
3123
3124    rule 'Place without multiset'
3125        match with
3126            m565:
3127                    any SATEL.APN.apnmm::Place
3128            m566:
3129                    not SATEL.APN.multisetmm::Multiset
3130
```

```
3131            subject to
3132                m565 !-(ownedPlaceMultiset)-> m566
3133
3134
3135        apply
3136            m567:
3137                    mprologTermReference::Term(
3138                        self = 'PlaceOwnedTerm'
3139                    )
3140            m568:
3141                    mprologTermReference::List
3142            subject to
3143                m567 --(nextTerm)-> m568
3144
3145        restrictions
3146            m567 derived from m565
3147
3148    end rule
3149
3150 end def
```

A.9 Excerto da primeira abordagem de implementação

A.9.1 Classe AxiomGraphProcessor

```
1  package SATEHALLOperationalSemantics;
2
3  import java.io.FileNotFoundException;
4
5  import java.lang.reflect.InvocationTargetException;
6  import java.util.HashMap;
7  import java.util.HashSet;
8  import java.util.LinkedList;
9  import java.util.List;
10
11
12 import modelOperationalSemantics.LoadContext;
13 import emfInterpreter.EMFLoader;
14 import emfInterpreter.instance.InstanceEntity;
15
16 public class AxiomGraphProcessor {
17
18     private EMFLoader loader;
19     private DatabaseOperations dbOp;
20     //<Axioma, Lista de Intenções
21     private HashMap<InstanceEntity, HashSet<String>> axiomsIntentionsinCond;
22     private HashMap<InstanceEntity, String> axiomsIntentionsinIncl;
23     private LinkedList<InstanceEntity> axioms;
24     private AxiomsInfoDatabase axiomsdatabase;
25
26     private TableGenerator tablegenerator;
27
28     public void perform() throws FileNotFoundException, SecurityException,
           IllegalArgumentException, ClassNotFoundException, NoSuchFieldException,
           IllegalAccessException, NoSuchMethodException, InvocationTargetException {
29
30         loader = new EMFLoader(new LoadContext().getResourceSet());
31         loader.loadMetaModel("./metamodel/SATEHALL/SATEHALL.ecore",true);
32         if(!loader.loadDatabase("SATEL.SATELPackage", "./model/SATEHALL/clockexample.xmi"))
33             return;
34         dbOp =  new DatabaseOperations(loader.getDatabase());
35
36         //Assume that there is only one model
37         InstanceEntity model = dbOp.getAllInstancesByName("Model").get(0);
38
39         for(InstanceEntity tim:dbOp.getEntitiesByRelName(model, "testIntentionModule")) {
40             axiomsdatabase = new AxiomsInfoDatabase();
41             tablegenerator = new TableGenerator(dbOp,axiomsdatabase);
42             tablegenerator.initTables(tim);
43             axiomsIntentionsinIncl = new HashMap<InstanceEntity, String>();
44             axiomsIntentionsinCond = new HashMap<InstanceEntity, HashSet<String>>();
45             axioms = (LinkedList<InstanceEntity>) dbOp.getAxiomsFromTestIntentionModule(tim);
46             getAxiomsIntentions();
47             processAxiom();
48         }
49     }
```

```
50
51
52      /*Obtain all predecessors and successors*/
53      private void getAxiomsIntentions() {
54          for(InstanceEntity axiom : axioms) {
55              AxiomInfo axiominfo = dbOp.getAxiomInfo(axiom);
56              this.axiomsdatabase.addAxiomInfo(axiom, axiominfo);
57              axiomsIntentionsinCond.put(axiom, dbOp.getIntentionsInCondition(axiom));
58              axiomsIntentionsinIncl.put(axiom, axiominfo.getIntention());
59          }
60      }
61
62      //buscar os poços em cada ponto
63      private List<InstanceEntity> getSinksAxioms() {
64          List<InstanceEntity> selectedAxioms= new LinkedList<InstanceEntity>();
65          for (InstanceEntity axiom : axioms)
66              if (axiomsIntentionsinCond.get(axiom).isEmpty())
67                  selectedAxioms.add(axiom);
68              else{
69                  HashSet<String> condIntentions = axiomsIntentionsinCond.get(axiom);
70                  String inclusion = axiomsIntentionsinIncl.get(axiom);
71                  //rever esta condição: se estiver na
72                  if (condIntentions.size()==1 && condIntentions.contains(inclusion) //se e
                        recursivo
73                          && noOtherAxiomHasInclusionInIntention(axiom,inclusion))
74                      selectedAxioms.add(axiom);
75              }
76          return selectedAxioms;
77      }
78
79      //retirar das condições se ja foram tratados os outros
80      private void updateAxiomDependencies(String intention) {
81              for (InstanceEntity axiom : axioms){
82                  axiomsIntentionsinCond.get(axiom).remove(intention);
83                  System.out.println("Agora tem: "+axiomsIntentionsinCond.get(axiom));
84              }
85      }
86
87      private boolean noOtherAxiomHasInclusionInIntention(InstanceEntity axtest, String
            intention) {
88          for (InstanceEntity axiom : axioms)
89              if (axtest!=axiom && axiomsIntentionsinIncl.get(axiom).equals(intention))
90                  return false;
91          return true;
92      }
93
94      private void updateAxiomForAllSelectedAxiom(List<InstanceEntity> selectedAxioms) {
95          for (InstanceEntity selAxiom: selectedAxioms) {
96              String intention = axiomsIntentionsinIncl.get(selAxiom);
97              if (noOtherAxiomHasInclusionInIntention(selAxiom, intention))
98                  updateAxiomDependencies(intention);
99          }
100     }
101
102     public void processAxiom() {
103         List<InstanceEntity> selectedAxioms = getSinksAxioms();
104         System.out.println("axiomas: "+axioms);
```

```
105         System.out.println("intencoesnascondicoes: "+axiomsIntentionsinCond);
106         while (!selectedAxioms.isEmpty()) {
107             System.out.println(selectedAxioms);
108             //gerar as tabelas;
109             tablegenerator.generateTables(selectedAxioms);
110
111             updateAxiomForAllSelectedAxiom(selectedAxioms);
112             axioms.removeAll(selectedAxioms);
113             selectedAxioms = getSinksAxioms();
114         }
115     }
116
117     public static void main(String[] args) throws FileNotFoundException, SecurityException,
            IllegalArgumentException, ClassNotFoundException, NoSuchFieldException,
            IllegalAccessException, NoSuchMethodException, InvocationTargetException {
118         new AxiomGraphProcessor().perform();
119     }
120
121 }
```

A.9.2 Classe AxiomInfo

```
1  package SATEHALLOperationalSemantics;
2
3  import java.util.HashMap;
4  import java.util.HashSet;
5  import java.util.LinkedList;
6
7  import emfInterpreter.instance.InstanceEntity;
8
9  class AxiomInfo {
10     private LinkedList<String> recVars;
11     private LinkedList<String> normVars;
12     private String intention;
13     private HashMap<String, String> intentionOfVars;
14     private InstanceEntity pattern;
15     private LinkedList<InstanceEntity> conditionAtoms;
16
17     public AxiomInfo(LinkedList<String> recVars, LinkedList<String> normVars,
18                      String intention, HashMap<String, String> intentionOfVars,
19                      InstanceEntity pattern, LinkedList<InstanceEntity> conditionAtoms) {
20         this.recVars = recVars;
21         this.normVars = normVars;
22         this.intention = intention;
23         this.intentionOfVars=intentionOfVars;
24         this.pattern = pattern;
25         this.conditionAtoms = conditionAtoms;
26     }
27
28
29     public InstanceEntity getPattern() {
30         return pattern;
31     }
32
33     public void setPattern(InstanceEntity pattern) {
```

```
34          this.pattern = pattern;
35      }
36
37      public LinkedList<InstanceEntity> getConditionAtoms() {
38          return conditionAtoms;
39      }
40
41      public void setConditionAtoms(LinkedList<InstanceEntity> conditionAtoms) {
42          this.conditionAtoms = conditionAtoms;
43      }
44
45      public void setIntention(String intention) {
46          this.intention = intention;
47      }
48
49      public HashSet<String> getSetOfIntentionsInCondition() {
50          return null;
51      }
52
53      public String getIntention(){
54          return intention;
55      }
56
57      public LinkedList<String> getRecVars() {
58          return recVars;
59      }
60
61      public void setRecVars(LinkedList<String> recVars) {
62          this.recVars = recVars;
63      }
64
65      public LinkedList<String> getNormVars() {
66          return normVars;
67      }
68
69      public void setNormVars(LinkedList<String> normVars) {
70          this.normVars = normVars;
71      }
72
73      public HashMap<String, String> getIntentionOfVars() {
74          return intentionOfVars;
75      }
76
77      public void setIntentionOfVars(HashMap<String, String> intentionOfVars) {
78          this.intentionOfVars = intentionOfVars;
79      }
80
81      public String getVarIntention(String var) {
82          return intentionOfVars.get(var);
83      }
84  }
```

A.9.3 Classe AxiomsInfoDatabase

```
1  package SATEHALLOperationalSemantics;
```

```
2
3  import java.util.HashMap;
4  import java.util.LinkedList;
5  import emfInterpreter.instance.InstanceEntity;
6
7  public class AxiomsInfoDatabase {
8
9      private HashMap<InstanceEntity, AxiomInfo> axiomsInfo;
10
11     public AxiomsInfoDatabase() {
12         axiomsInfo = new HashMap<InstanceEntity, AxiomInfo>();
13     }
14
15     public AxiomInfo getAxiomInfo(InstanceEntity axiom) {
16         return axiomsInfo.get(axiom);
17     }
18
19     public void addAxiomInfo(InstanceEntity axiom, AxiomInfo info) {
20         axiomsInfo.put(axiom, info);
21     }
22     public InstanceEntity cloneAxiom(InstanceEntity axiom) {
23         return null;
24     }
25
26     public LinkedList<String> getRecursiveVars(InstanceEntity axiom) {
27         return axiomsInfo.get(axiom).getRecVars();
28     }
29
30     public LinkedList<String> getNormalVars(InstanceEntity axiom){
31         return axiomsInfo.get(axiom).getNormVars();
32     }
33
34     public String getIncludedInclusion(InstanceEntity axiom) {
35         return axiomsInfo.get(axiom).getIntention();
36     }
37     public LinkedList<InstanceEntity> getConditionAtoms(InstanceEntity axiom) {
38         return axiomsInfo.get(axiom).getConditionAtoms();
39     }
40 }
```

A.9.4 Classe Bounds

```
1  package SATEHALLOperationalSemantics;
2
3  /**
4   * Bounds stores the defined range for number of events and depth
5   * @note data member are public to reduce the number of method calls and
6   *       to agilize the work with this kind of boundaries
7   * @author Rui Domingues
8   */
9
10 public class Bounds {
11     public int minDepth;
12     public int minNbEvents;
13     public int maxDepth;
```

```
14      public int maxNbEvents;
15
16      public Bounds() {
17          minDepth = Integer.MAX_VALUE;
18          maxDepth = Integer.MIN_VALUE;
19          minNbEvents = Integer.MAX_VALUE;
20          maxNbEvents = Integer.MIN_VALUE;
21      }
22
23      public Bounds(int minDepth, int minNbEvents, int maxDepth, int maxNbEvents) {
24          this.minDepth = minDepth;
25          this.minNbEvents = minNbEvents;
26          this.maxDepth = maxDepth;
27          this.maxNbEvents = maxNbEvents;
28      }
29
30  }
```

A.9.5 Classe DatabaseOperations

```
1   package SATEHALLOperationalSemantics;
2
3   import java.lang.reflect.InvocationTargetException;
4   import java.util.HashMap;
5   import java.util.HashSet;
6   import java.util.LinkedList;
7
8   import java.util.List;
9
10
11  import emfInterpreter.instance.InstanceAttribute;
12  import emfInterpreter.instance.InstanceDatabase;
13  import emfInterpreter.instance.InstanceEntity;
14  import emfInterpreter.instance.InstanceRelation;
15
16  public class DatabaseOperations {
17
18      private InstanceDatabase idb;
19
20      public DatabaseOperations(InstanceDatabase idb) {
21          this.idb = idb;
22      }
23
24      public InstanceDatabase getDatabase(){
25          return idb;
26      }
27
28      //pôr noutro sitio
29      //foi mudada a visibiliade de cloneEntity
30      //
31      public InstanceEntity cloneEntityTree(InstanceRelation irparent, InstanceEntity entity)
            throws SecurityException, IllegalArgumentException, ClassNotFoundException,
            NoSuchMethodException, IllegalAccessException, InvocationTargetException,
            NoSuchFieldException{
32          InstanceEntity cloned = idb.cloneEntity(entity);
```

```
33          idb.getLoadedClasses().add(cloned);
34          if (irparent!=null) {
35              irparent.setTarget(cloned);
36              idb.getLoadedRelations().add(irparent);
37          }
38          for (InstanceRelation oldir: idb.getRelationsByInstanceEntity(entity))
39              cloneEntityTree(new InstanceRelation(cloned,oldir.getMetaRelation(),null), oldir.
                    getTarget());
40          return cloned;
41      }
42
43
44      public List<InstanceEntity> getRelatedEntitiesByInstanceEntity(InstanceEntity ie) {
45          List<InstanceEntity> ilist = new LinkedList<InstanceEntity>();
46          for(InstanceRelation ir : idb.getLoadedRelations()) {
47              if(ir.getSource() == ie)
48                  ilist.add(ir.getTarget());
49          }
50          return ilist;
51      }
52
53      public List<InstanceEntity> getEntitiesByRelName(InstanceEntity ie, String relname) {
54          LinkedList<InstanceEntity> resultlist = new LinkedList<InstanceEntity>();
55          for(InstanceRelation ir: idb.getRelationsByInstanceEntity(ie))
56              if(ir.getMetaRelation().getName().equals(relname))
57                  resultlist.add(ir.getTarget());
58          return resultlist;
59      }
60
61
62      //get an attribute of a classe
63      public Object getAttrByName(InstanceEntity ie, String relname) {
64          for(InstanceAttribute ia: idb.getAttributesByInstanceEntity(ie))
65              if(ia.getMetaAttribute().getName().equals(relname))
66                  return ia.getValue();
67          return null;
68      }
69      //Tirar todas as instancias do tipo name
70      public List<InstanceEntity> getAllInstancesByName(String name) {
71          List<InstanceEntity> result=new LinkedList<InstanceEntity>();;
72          for(InstanceEntity ie:idb.getLoadedClasses())
73              if(ie.getMetaEntity().getName().equals(name))
74                  result.add(ie);
75
76          return result;
77      }
78
79      //Obter uma lista de todos os condition atoms existentes
80      public LinkedList<InstanceEntity> getAxiomConditionsAtom(InstanceEntity axiom) {
81          LinkedList<InstanceEntity> conditions=new LinkedList<InstanceEntity>();
82          LinkedList<InstanceEntity> condList = (LinkedList<InstanceEntity>)
                    getEntitiesByRelName(axiom, "condition");
83          if (condList.isEmpty()) return conditions;
84          List<InstanceEntity> cbList = getEntitiesByRelName(condList.get(0), "conditionBody");
85          if (cbList.isEmpty()) return conditions;
86          return (LinkedList<InstanceEntity>) getEntitiesByRelName(cbList.get(0), "conditionAtom
                    ");
```

```
87      }
88
89      public InstanceEntity getAxiomInclusion(InstanceEntity axiom) {
90          return getEntitiesByRelName(axiom, "inclusion").get(0);
91      }
92
93
94      public InstanceEntity gethmlTermFromInclusion(InstanceEntity inclusion) {
95          return getEntitiesByRelName(inclusion, "hmlTerm").get(0);
96      }
97
98      public InstanceEntity getHmlTermFromAxiom(InstanceEntity axiom) {
99          return gethmlTermFromInclusion(getAxiomInclusion(axiom));
100     }
101
102     public List<String> getVarNamesFromHMLTerm(InstanceEntity hmlTerm/*:HMLTerm*/) {
103
104         List<String> varnames=new LinkedList<String>();
105         for (InstanceEntity hterm: getEntitiesByRelName(hmlTerm, "hmlTerm") )
106                 for (InstanceEntity hmlVar: getEntitiesByRelName(hterm, "hmlVariable") )
107                     varnames.add((String) getAttrByName(hmlVar, "name"));
108         return varnames;
109     }
110
111     public List<String> getVarNamesFromHMLFormula(InstanceEntity hmlFormula) {
112
113         List<String> varnames=new LinkedList<String>();
114                 for (InstanceEntity hmlVar: getEntitiesByRelName(hmlFormula, "hmlVariable") )
115                     varnames.add((String) getAttrByName(hmlVar, "name"));
116         return varnames;
117     }
118
119     public String getVarDecName(InstanceEntity hmlvardec) {
120         return  (String)getAttrByName(hmlvardec, "name");
121     }
122
123
124     //devolve todas as variaveis presentes numa inclusao
125     public List<String> getVarsFromInclusion(InstanceEntity inclusion) {
126         return this.getVarNamesFromHMLTerm(getEntitiesByRelName(inclusion,"hmlTerm").get(0));
127     }
128
129     //Varios conditionAtoms sao um inclusion com varias variaveis
130     public List<Pair<String,List<String>>> getVarsFromAllConditionAtoms(List<InstanceEntity>
            condAt) {
131         List<Pair<String, List<String>>> result = new LinkedList<Pair<String,List<String>>>();
132         for (InstanceEntity cAt : condAt)
133             if (cAt.getMetaEntity().getName().equals("Inclusion"))
134                 result.add(new Pair(getInclusionName(cAt), getVarsFromInclusion(cAt) ));
135
136         return result;
137     }
138     public HashMap<String, String> getVarsAndTheirIntention(InstanceEntity axiom) {
139         HashMap<String, String> result=new HashMap<String, String>();
140         for (Pair<String,List<String>> intvars : getVarsFromAllConditionAtoms(this.
                getAxiomConditionsAtom(axiom)))
141             for (String var : intvars.snd)
```

```
142                 result.put(var, intvars.fst);
143             return result;
144         }
145
146         //devolve o conjunto de intençoes do axioma presentes nas condicoes
147         public HashSet<String> getIntentionsInCondition(InstanceEntity axiom) {
148             HashSet<String> intentions=new HashSet<String>();
149             for (InstanceEntity cAt: getAxiomConditionsAtom(axiom) )
150                 if (cAt.getMetaEntity().getName().equals("Inclusion"))
151                     intentions.add(getInclusionName(cAt));
152                     //obter as variaveis associadas
153             return intentions;
154         }
155
156         //Dada uma inclusao devolve o seu nome
157         public String getInclusionName(InstanceEntity inclusion) {
158             return (String)getAttrByName(getEntitiesByRelName(inclusion, "in").get(0),"name");
159         }
160
161         /*Devolve todos os axiomas presentes num testintentionModule
162          */
163         public List<InstanceEntity> getAxiomsFromTestIntentionModule(InstanceEntity tim) {
164             InstanceEntity tib = getEntitiesByRelName(tim, "testIntentionBody").get(0);
165             InstanceEntity  axDecl = getEntitiesByRelName(tib, "axiomDeclaration").get(0);
166             return   getEntitiesByRelName(axDecl, "axiom");
167         }
168
169         /*Dado um axioma, devolve todas as suas
170          * variaveis recursivas e normais
171          */
172         public Pair<List<String>,List<String>> getVars(InstanceEntity axiom) {
173             List<String> recvars = new LinkedList<String>();
174             List<String> normvars = new LinkedList<String>();
175             InstanceEntity inclusion = getAxiomInclusion(axiom);
176             List<String> varInInclusion = getVarsFromInclusion(inclusion);
177             String incIntName = (String) getInclusionName(inclusion);
178             List<Pair<String,List<String>>> cintNameVars = getVarsFromAllConditionAtoms(
                    getAxiomConditionsAtom(axiom));
179             for (String varIncl:varInInclusion)
180                 for (Pair<String,List<String>> pair : cintNameVars)
181                     if (pair.fst.equals(incIntName))
182                         for (String varC: pair.snd) {
183                             if (varC.equals(varIncl))
184                                     recvars.add(varIncl);
185                             else normvars.add(varC);
186                         }
187                     else for (String varC: pair.snd)
188                                 normvars.add(varC);
189             return new Pair<List<String>,List<String>>(recvars, normvars);
190         }
191
192         public AxiomInfo getAxiomInfo(InstanceEntity axiom ) {
193             Pair<List<String>, List<String>> vars = getVars(axiom);
194             return new AxiomInfo(
195                     (LinkedList<String>)vars.fst,
196                     (LinkedList<String>)vars.snd,
197                     this.getInclusionName( this.getAxiomInclusion(axiom)),
```

```
198                 this.getVarsAndTheirIntention(axiom),
199                 this.getHmlTermFromAxiom(axiom),
200                 this.getAxiomConditionsAtom(axiom)
201         );
202 
203     }
204 
205 }
```

A.9.6 Classe HMLFormOper

```
1  package SATEHALLOperationalSemantics;
2  
3  import emfInterpreter.instance.InstanceEntity;
4  import emfInterpreter.instance.InstanceRelation;
5  
6  public class HMLFormOperWrapper {
7  
8      public static int countEvents(InstanceEntity hmlformula, DatabaseOperations dbOp) {
9          if (hmlformula.getMetaEntity().getName().equals("BOPAnd"))
10             return countEvents(dbOp.getEntitiesByRelName(hmlformula, "booleanTermL").get(0),
                   dbOp)+
11                             countEvents(dbOp.getEntitiesByRelName(hmlformula, "booleanTermR").
                                   get(0),dbOp);
12 
13         else if (hmlformula.getMetaEntity().getName().equals("HMLEvent"))
14                return 1; //to do:
15 
16         else {
17             int sum=0;
18             for (InstanceEntity ie: dbOp.getRelatedEntitiesByInstanceEntity(hmlformula))
19                 sum+=countEvents(ie, dbOp);
20             return sum;
21         }
22     }
23 
24     public static int countDepth(InstanceEntity hmlformula, DatabaseOperations dbOp) {
25         if (hmlformula.getMetaEntity().getName().equals("BOPAnd"))
26             return Math.max(countDepth(dbOp.getEntitiesByRelName(hmlformula, "booleanTermL").
                   get(0), dbOp),
27                             countDepth(dbOp.getEntitiesByRelName(hmlformula, "booleanTermR").
                                   get(0),dbOp));
28         else if (hmlformula.getMetaEntity().getName().equals("HMLEvent"))
29                 return 1;
30         else {
31             int sum=0;
32             for (InstanceEntity ie: dbOp.getRelatedEntitiesByInstanceEntity(hmlformula))
33                 sum+=countDepth(ie, dbOp);
34             return sum;
35         }
36     }
37 
38     public static InstanceEntity clone(InstanceEntity hmlformula, DatabaseOperations dbOp) {
39         return null;
40     }
```

```
41
42      public static void subst(InstanceRelation relationparent,
43              InstanceEntity hmlFormula, String variable,
44              InstanceEntity newVal, DatabaseOperations dbOp) {
45
46          if (hmlFormula.getMetaEntity().getName().equals("VariableDec")) {
47              if (dbOp.getVarDecName(hmlFormula).equals(variable))
48                  relationparent.setTarget(newVal);
49              } else
50                  for (InstanceRelation ir: dbOp.getDatabase().getRelationsByInstanceEntity(
                        hmlFormula))
51                      subst(ir,ir.getTarget(),variable,newVal, dbOp);
52      }
53  }
```

A.9.7 Classe Table

```
1   package SATEHALLOperationalSemantics;
2
3   import java.util.ArrayList;
4   import java.util.HashSet;
5
6   public class Table {
7
8       private String intentionName;
9       private ArrayList<Tuple> tuples;
10      private Bounds bounds;
11      private HashSet<String> tuplesToCheck; //forma rapida de verificar a existencia de um tuplo similar
12      //implementar um método no tuplo que permita obter um tuplo stringified
13
14      public Table(String intentionName, int maxSize) {
15          this.intentionName = intentionName;
16          tuples = new ArrayList<Tuple>(maxSize);
17          bounds = new Bounds();
18      }
19
20      public void addRow(Tuple tuple) {
21          tuples.add(tuple);
22          updateNbEvents(tuple.getFixedNbEvents());
23          updateDepth(tuple.getFixedDepth());
24      }
25      public int getNbOfTuples(){
26          return tuples.size();
27      }
28
29      public String getIntentionName() {
30          return intentionName;
31      }
32
33      private void updateNbEvents(int value) {
34          if (value > bounds.maxNbEvents)
35              bounds.maxNbEvents = value;
36          if (value < bounds.minNbEvents)
37              bounds.minNbEvents = value;
38      }
```

```
39      private void updateDepth(int value) {
40          if (value > bounds.maxDepth)
41              bounds.maxDepth = value;
42          if (value < bounds.minDepth)
43              bounds.minDepth = value;
44      }
45
46      public Bounds getBounds() {
47          return bounds;
48      }
49
50
51      @Override
52      public String toString() {
53          String result = "Tabela da intencao " + intentionName + "\n\n";
54          for (Tuple tup : tuples)
55              result+=tup+"\n";
56          return result;
57      }
58  }
```

A.9.8 Classe TableGenerator

```
1   package SATEHALLOperationalSemantics;
2
3   import java.util.HashMap;
4   import java.util.LinkedList;
5   import java.util.List;
6
7   import emfInterpreter.instance.InstanceEntity;
8
9   public class TableGenerator {
10      final static public int MAXAXIOMSOL=10;
11
12      private DatabaseOperations dbOp;
13      private TableSet tables;
14      private AxiomsInfoDatabase axiomsInfoDb;
15
16      public TableGenerator(DatabaseOperations dbOp, AxiomsInfoDatabase axiomsInfoDb) {
17          this.dbOp=dbOp;
18          this.axiomsInfoDb = axiomsInfoDb;
19      }
20
21      /**
22       * iniciar as tabelas de um intentionModule
23       * @param intentionModule
24       */
25      public  void initTables(InstanceEntity intentionModule) {
26          try {
27              InstanceEntity tii = dbOp.getEntitiesByRelName(intentionModule, "
                    testIntentionInterface").get(0);
28              for (InstanceEntity intention : dbOp.getEntitiesByRelName(tii, "intention")){
29                  String name = (String) dbOp.getAttrByName(intention, "name");
30                  tables.addEmptyTable(name);
31              }
```

```
32          } catch(Exception e) {
33              System.out.println("Unable to create tables! Define interface!");
34          }
35      }
36
37
38
39
40
41      public Tuple processRow(Tuple line, AxiomInfo axiominfo) {
42          HashMap<String, Bounds> tuplevarsRanges = new HashMap<String,  Bounds>();
43          HashMap<String, Table> varRes= new HashMap<String,Table>();
44          List<InstanceEntity> conds = line.getConditions();
45          HashMap<String, Bounds> bounds = ConditionsOperationWrapper.inferBounds(conds, dbOp,
                axiominfo.getNormVars());
46          //Como retirar as condicoes????
47
48          for (String var: axiominfo.getNormVars()) {
49                  String intInVar = axiominfo.getIntentionOfVars().get(var);
50                  Table table= tables.pick(intInVar);
51                  varRes.put(var, table);
52                  tuplevarsRanges.put(var, calcInterval(table, bounds.get(var)));
53          }
54
55          return new Tuple(conds, line.getPattern(),
56                                  HMLFormOperWrapper.countEvents(line.getPattern(), dbOp),
57                                  HMLFormOperWrapper.countDepth(line.getPattern() ,dbOp),
58                                  tuplevarsRanges, varRes );
59      }
60
61      private Bounds calcInterval(Table intTable, Bounds varbounds) {
62          return new Bounds(
63                  Math.max(varbounds.minNbEvents, intTable.getBounds().minNbEvents),
64                  Math.min(varbounds.maxNbEvents, intTable.getBounds().maxNbEvents),
65                  Math.max(varbounds.minDepth, intTable.getBounds().minDepth),
66                  Math.min(varbounds.maxDepth, intTable.getBounds().maxDepth));
67      }
68
69
70      public void generateTables(List<InstanceEntity> selectedAxioms) {
71          for (InstanceEntity selAx : selectedAxioms) {
72              AxiomInfo axiominfo = axiomsInfoDb.getAxiomInfo(selAx);
73              String intName = axiominfo.getIntention();
74
75              Table table = tables.pick(intName);
76              int oldtablesize=-1; // para condicao de paragem quando deixam de ser adicionadas mais rows
77              if (!axiominfo.getRecVars().isEmpty()) {
78                      while (true) {
79                          if (table.getNbOfTuples()> MAXAXIOMSOL || oldtablesize==table.
                                getNbOfTuples())
80                              break;
81                          oldtablesize = table.getNbOfTuples();
82                      //varSolutions=solve(table, recVars, selAx.getCond()); // devolver List<Pair<String,List<
83                      //varSolProduct = crossVars(varSolutions);
84                      //forall sol in varSolProduct
85                      //Tuple line = subs(sol,x);
86                      // line = processLine
```

```
87                      //table.addRow(line);
88                  }
89              } else {
90                  Tuple line = new Tuple(axiominfo.getConditionAtoms(), axiominfo.getPattern());
91                  line = processRow(line, axiominfo);
92                  table.addRow(line);
93              }
94          }
95
96
97      }
98  }
```

A.9.9 Classe Tuple

```
1   package SATEHALLOperationalSemantics;
2
3   import java.util.HashMap;
4   import java.util.List;
5
6   import emfInterpreter.instance.InstanceEntity;
7
8   public class Tuple {
9       private List<InstanceEntity> conditions;
10      private InstanceEntity pattern; //HMLTerm
11      private int fixedDepth;
12      private int fixednbEvents;
13      //<VAR, BOUNDS | Table>
14      private HashMap<String,Bounds> varsBounds;
15      private HashMap<String,Table> varRes;
16
17      public List<InstanceEntity> getConditions() {
18          return conditions;
19      }
20
21      public InstanceEntity getPattern() {
22          return pattern;
23      }
24
25      public void setPattern(InstanceEntity pattern) {
26          this.pattern = pattern;
27      }
28
29      public HashMap<String, Bounds> getVarsBounds() {
30          return varsBounds;
31      }
32
33      public Bounds getVarBounds(String var) {
34          return this.varsBounds.get(var);
35      }
36
37      public HashMap<String, Table> getVarRes() {
38          return varRes;
39      }
40
```

```
41      public int getFixedDepth() {
42          return fixedDepth;
43      }
44      public int getFixedNbEvents() {
45          return fixednbEvents;
46      }
47
48      public Tuple(List<InstanceEntity> condAtoms, InstanceEntity pattern,
49                  int fixedNbEvents, int fixedDepth, HashMap<String,Bounds> varsbounds, HashMap
                        <String,Table> varRes) {
50          this.conditions=condAtoms;
51          this.pattern=pattern;
52          this.fixednbEvents = fixedNbEvents;
53          this.fixedDepth = fixedDepth;
54          this.varsBounds = varsbounds;
55          this.varRes = varRes;
56      }
57
58      public Tuple(List<InstanceEntity> condAtoms, InstanceEntity pattern) {
59              this.conditions=condAtoms;
60              this.pattern=pattern;
61
62              //this.varsBounds = new HashMap<String, Bounds>();
63              //this.varRes = null;
64      }
65
66      @Override
67      public String toString() {
68          //As condicoes
69          //o padrao
70          //fixedvalues
71          //as vars e bounds
72          //var res
73          return "Tuplo desc";
74      }
75  }
```

Bibliografia

[1] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M. Küster. Analysis of model transformations via alloy.

[2] Francisco R. de Andrade, João P. Faria, C. R. Ana Paiva, and Antónia Lopes. Geração de testes a partir de especificações algébricas de tipos genéricos usando alloy. INFORUM '11, Lisbon, 2011.

[3] Ana Barbosa, Ana C.R. Paiva, and José Creissac Campos. Test case generation from mutated task models. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, pages 175–184, New York, NY, USA, 2011. ACM.

[4] Bruno Barroca, Levi Lucio, Vasco Amaral, Vasco Sousa, and Roberto Felix. Dsltrans: A turing incomplete transformation language. In *Proc. 3rd International Conference on Software Languages Engineering - SLE 2010*. Springer-Verlag, 2010.

[5] Bruno Barroca, Levi Lucio, Didier Buchs, Vasco Amaral, and Luis Pedro. Dsl composition for model-based test generation. *ECEASST*, 21, 2009.

[6] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[7] Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneve, 1997.

[8] Huo Yan Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 7:250–295, July 1998.

[9] Li Dan and Bernhard K. Aichernig. Combining algebraic and model-based test case generation. In *In Proceedings of First International Colloquium on Theoretical Aspects of Computing*. SpringerVerlag.

[10] A. J. J. Dick and Phil Watson. Order-sorted term rewriting. *Comput. J.*, 34(1):16–19, 1991.

[11] Roong-Ko Doong and Phyllis G. Frankl. The astoot approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3:101–130, April 1994.

[12] Website:
http://www.goldpractices.com/practices/mbt/.

[13] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[14] Merlin Hughes and David Stotts. Daistish: Systematic algebraic testing for oo programs in the presence of side-effects. In *IN PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND VERIFICATION*, pages 53–61. ACM, 1996.

[15] Pankaj Jalote. *An integrated approach to software engineering*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[16] Liang Kong, Hong Zhu, and Bin Zhou. Automated testing ejb components based on algebraic specifications. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 717–722, Washington, DC, USA, 2007. IEEE Computer Society.

[17] Levi Lúcio. *SATEL — A Test Intention Language for Object-Oriented Specifications of Reactive Systems*. PhD thesis, Université de Genève - Switzerland, 2009.

[18] Levi Lucio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In *ACM/IEEE MoDELS 2010*. Springer-Verlag, 10 2010.

[19] Williams M., Succi G., and Marchesi L. Black box testing. In *Traditional and Agile Software Engineering*, ISSTA '94. Addison-Wesley, 2003.

[20] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[21] Dennis Peters and David L. Parnas. Generating a test oracle from program documentation: work in progress. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 58–65, New York, NY, USA, 1994. ACM.

[22] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80(1):1–34, March 1991.

[23] Harry Robinson. Finite state model-based testing on a shoestring. Star West, 1999.

[24] Colin Stirling. *Modal and temporal properties of processes*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[25] Mark Utting. *Practical model-based testing : a tools approach*. Morgan Kaufmann Publishers, Amsterdam Boston, 2007.

[26] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Formal methods and testing. chapter Model-based testing of object-oriented reactive systems with spec explorer, pages 39–76. Springer-Verlag, Berlin, Heidelberg, 2008.

[27] Bo Yu, Liang Kong, Yufeng Zhang, and Hong Zhu. Testing java components based on algebraic specifications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 190–199, Washington, DC, USA, 2008. IEEE Computer Society.