Introdução a Métodos Formais: Especificação, Semântica e Verificação de Sistemas Concorrentes

David Déharbe*

Anamaria M. Moreira*

Leila Ribeiro**

Vanderlei Moraes Rodrigues**

Resumo

Este tutorial apresenta uma visão geral de métodos formais para a especificação, semântica e verificação de sistemas concorrentes. Um método de especificação formal dá uma descrição precisa de um sistema em uma notação com uma sintaxe e semântica bem definidas. Esta semântica associa um modelo matemático ao sistema que pode então ser analisado usando técnicas de verificação formal. São discutidos diversos métodos de especificação formal e apresenta-se com mais detalhes o método de gramáticas de grafos. Para introduzir os modelos semânticos, utiliza-se sistemas de transição (uma método clássico de descrição de sistemas seqüenciais ou concorrentes) e apresenta-se uma classificação destes modelos. Introduz-se as principais abordagens de verificação formal e discute-se o método de verificação de modelos. Para ilustrar todos os métodos e modelos, emprega-se como exemplo um sistema de controle de trens.

*Departamento de Informática e Matemática Aplicada,
Universidade Federal do Rio Grande do Norte
e-mail: {david,anamaria}@dimap.ufrn.br
**Instituto de Informática,
Universidade Federal do Rio Grande do Sul
e-mail: {leila,vandi}@inf.ufrgs.br

Abstract

This tutorial is an overview of formal methods for specification, semantics, and verification of concurrent systems. A formal specification method gives a precise description of a computational system in a notation with a well-defined syntax and semantics. This semantics associates a mathematical model to the system being described, and formal verification techniques may be used to analyze properties of this model. We consider several specification methods, and present in further detail the graph grammars method. To introduce the semantic mathematical models, we use transition systems (a standard framework to describe sequential and concurrent systems) and give a classification of semantic models. This tutorial introduces the main approaches to formal verification, and discusses the model checking method. The example illustrating all concepts and methods is a control system for a train network.

Keywords: Formal specification, formal semantics, formal verification, graph grammars, model checking

Sumário

1. Introdução

Com o crescimento do tamanho e complexidade dos sistemas de software, surgiu a necessidade de construir especificações mais precisas para descrever o comportamento desses sistemas (semântica), bem como possibilitar a verificação de propriedades destes sistemas e da correção de suas implementações. A linguagem natural, que até então tinha sido o meio mais utilizado para especificar sistemas, normalmente gera documentos bastante ambíguos, incompletos e inconsistentes, e não permite termos a certeza de que a especificação exibe as propriedades necessárias, e nem que a implementação está correta. Mas, que linguagem podemos utilizar para especificar um sistema e permitir sua verificação? Para responder essa pergunta, vamos primeiro recordar uma parte das origens da matemática...

Uma maneira de se chegar a conclusões sobre propriedades de um sistema (não necessariamente computacional) é utilizar uma prova científica. Nestas provas parte-se de uma hipótese a ser comprovada, realiza-se experimentos para confirmar tal hipótese, faz-se previsões de novos resultados que são também testados com experimentos e, caso estes experimentos tenham comprovado a hipótese, esta é aceita como teoria. Um exemplo seria o teste de um programa. Tomemos como hipótese: *O programa sempre gera os resultados esperados.* Para confirmar tal hipótese, podemos fazer inúmeras baterias de testes. Caso o programa sempre se comporte da maneira esperada nos testes, concluímos que ele sempre irá funcionar. Mas, e se esquecemos de testar um caso e justamente para este caso o sistema não se comporta como esperado? Como normalmente existe um número infinito de casos de teste, esta situação é possível, senão provável. Neste exemplo, fica evidente uma fraqueza do método científico: uma teoria é apenas altamente provável, com base nas evidências disponíveis (sempre existe um elemento de dúvida). Como o método científico depende de observações e percepções (que são falíveis), uma teoria que é verdade indubitável em um dia pode ser comprovada como errada no dia seguinte. Em várias aplicações de computadores, como controles de foguetes, armas e usinas nucleares, contas bancárias, etc., não podemos simplesmente *pensar* que o programa correspondente está correto, precisamos *ter certeza* disto. Mas como?

Na Grécia antiga (VI a. C.), Pitágoras inventou um método para provar que algumas asserções são verdadeiras sem ter que testá-las para todos os casos. Por exemplo, ele sabia que o famoso *Teorema de Pitágoras* era verdadeiro para todos os triângulos retângulos sem tê-lo testado em todos os (infinitos) casos. Como isto é possível? A idéia básica é que a realização de uma prova deve ser baseada apenas em fatos assumidos como verdadeiros (axiomas) e em um raciocínio lógico. Esta prova é chamada de *prova matemática*, e a conclusão é chamada de teorema. Ao contrário do método científico, uma prova matemática é completamente destituída de dúvida e não depende de evidências resultantes de experiências sujeitas a falhas. Um teorema, uma vez provado ser verdadeiro, será sempre verdadeiro. A veracidade do teorema segue

apenas da validade das regras lógicas a axiomas empregas na prova e não depende de experimentação. Portanto, o método matemático pode ser utilizado para garantir que programas estão corretos ou apresentam certas propriedades.

Os métodos formais para computação estabelecem maneiras de construir e analisar sistemas com base em métodos matemáticos. Estes métodos podem ser utilizados para descrição de sistemas, verificação e análise de propriedades, verificação da correção de implementações, otimização, descrição e verificação do processo de desenvolvimento do sistema, como suporte à geração de bons conjuntos de casos de teste e até mesmo para orientar no encontro de soluções mais elegantes, simples e eficientes.

Este tutorial oferece uma introdução a métodos que podem ser utilizados para especificar sistemas de maneira precisa, bem como técnicas que podem ser utilizadas para realizar a verificação de propriedades desses sistemas. Esses métodos se chamam formais por utilizarem conceitos matemáticos. A especificação formal auxilia na construção de um modelo matemático que descreve o comportamento de um sistema (semântica), e esse modelo pode ser analisado através de técnicas de verificação formal.

É importante ressaltar, no entanto, que mesmo se a verificação formal é provavelmente a principal aplicação das especificações formais, e uma das ênfases desse tutorial, ela não é a única. Em alguns casos, uma especificação formal de partes de um sistema pode ser efetuada apenas com o intuito de garantir uma melhor compreensão de detalhes importantes do mesmo, por exemplo. Freqüentemente, o simples fato de tentar descrever formalmente uma característica do sistema pode evidenciar ambigüidades e mal-entendidos quanto aos requisitos correspondentes.

Como exemplo, consideremos a seguinte especificação para um procedimento de ordenação de uma lista de números (função `ordenar`):

> *A função* `ordenar` *recebe como argumento uma lista possivelmente não ordenada de números e a ordena em ordem crescente, ou seja, fornece como resultado uma lista onde cada elemento é superior ao precedente.*

Será que essa especificação corresponde ao que queremos exatamente? É fácil notar que não. Para começar, nada nos impede de implementar a função `ordenar` de maneira a que ela sempre me devolva uma lista "onde cada elemento é superior ao precedente", mas que não contém os mesmos números que a lista original. Essa seria uma interpretação possível para alguém que não entende a instrução "a ordena em ordem crescente" e se guia pela explicação acima. Esse erro de interpretação, é claro, nunca aconteceria em uma especificação tão simples (espera-se que todo programador saiba o que é ordenar uma lista...). Mas e se a especificação fizesse parte do controle de um robô cirurgião e estivesse falando de possibilidades de cortes (medianos, laterais, etc.), será que o programador teria o mesmo discernimento para completar não-ditos como o acima?

Mas voltando a nosso exemplo, a especificação abaixo nos garantiria o que quere-

mos?

> *A função* `ordenar` *recebe como argumento uma lista possivelmente não ordenada de números e a ordena em ordem crescente, ou seja, fornece como resultado uma lista contendo exatamente os mesmos elementos que a lista original redistribuídos de maneira que cada elemento é superior ao precedente.*

Parece bom, mas o que acontecerá caso a lista contenha repetições? Se quisermos satisfazer a condição "exatamente os mesmos elementos" as repetições devem ser preservadas na nova lista, mas isso nos impediria de satisfazer a condição "cada elemento é superior ao precedente". Pelo conceito usual de ordenação, é a segunda condição que está excessivamente forte. Deveria ser superior ou igual, mas na linguagem informal, freqüentemente utilizamos o termo superior para designar ambos, da mesma maneira que dificilmente distinguimos o OU do OU-exclusivo. Esse é o tipo de detalhe que passa despercebido quando escrevemos uma especificação informal e que normalmente é definido pelo implementador. No entanto, se tivéssemos escrito (semi)formalmente algo como a descrição abaixo, a escolha entre o símbolo de "superior" e o de "superior ou igual" teria sido efetuada em tempo de especificação, pois nesse contexto a ambigüidade não é tolerada.

> *Dada* `ListaOriginal` $=< e_1, e_2, ..., e_n >$,
> *Então* `ordenar(ListaOriginal)` = `ListaOrdenada` $=< e'_1, e'_2, ..., e'_n >$,
> *onde:*
>
> - $\forall e \in$ `ListaOriginal` : $e \in$ `ListaOrdenada`, *e*
> - $\forall e \in$ `ListaOrdenada` : $e \in$ `ListaOriginal`, *e*
> - $\forall e$: $\#(e,$ `ListaOrdenada`$) = \#(e,$ `ListaOriginal`$)$, *onde* $\#(e, L)$ *fornece o número de ocorrências do número* e *na lista* L, *e*
> - $\forall i, j \in 1..n : i > j \rightarrow e'_i \geq e'_j$

Adicionalmente, o fato de fazer um esforço adicional para traduzir os requisitos em uma forma matemática, faz com que naturalmente o especificador pense um pouco melhor no que está escrevendo, pois o que era ambíguo em linguagem natural (como a propriedade dos mesmos elementos desse exemplo) passa a ser errado quando explicitado formalmente. Por exemplo, a ausência das 3 primeiras condições na especificação formal da ordenação faria que quando a implementação viesse errada o especificador não teria como justificar-se.

Esse engano de considerar que só é interessante fazer uma especificação formal de um sistema se o objetivo é verificá-lo formalmente é comum e constitui um dos chamados "mitos" dos métodos formais. Alguns desses mitos foram primeiramente identificados por J. A. Hall em [Hal90] e em seguida complementados por J. P.

Bowen e M. C. Hinchey em [BH95]. Eles representam visões falsas que leigos e principalmente, e mais grave, profissionais de informática, têm sobre a aplicabilidade de técnicas formais. Diversos desses mitos pregam a dificuldade de se utilizar os métodos formais: alto custo, maior duração de projetos, necessidade de matemática de alto nível, impossibilidade de compreensão pelo usuário, falta de ferramentas, etc. No entanto, experiências bem sucedidas têm demonstrado a sua falta de fundamento. Projetos bem coordenados, com a aplicação das técnicas formais apropriadas nos pontos apropriados, com especificações bem comentadas, auxiliando o cliente a "ver" sua especificação através dos símbolos matemáticos, etc., têm sido completados com sucesso e duração e custos inferiores ao estimado [CW96].

1.1 Semântica Formal

Quando consideramos o mundo real, podemos analisar, quantificar e prever vários fenômenos físicos porque eles estão descritos através de equações matemáticas. O desenvolvimento desses modelos matemáticos foi de fundamental importância para várias descobertas da ciência moderna. Se considerarmos o *mundo virtual*, a construção de modelos se torna ainda mais importante devido a vários fatores, dentre os quais estão:

- *O mundo virtual, ao contrário do real, não existia, ele está sendo desenvolvido por nós.* Portanto, podemos usar os modelos para modificar e melhorar este universo (e não apenas para analisá-lo).
- *O universo virtual é bem menos tangível que o real.* Nós só conseguimos perceber esse universo através de estímulos obtidos nas interfaces que possuímos (vídeo, impressora, etc.). Portanto, temos uma visão extremamente limitada do que está ocorrendo "dentro do computador, ou de uma rede de computadores". Isso torna a análise de sistemas computacionais bastante complexa.

Ao contrário dos modelos do mundo real, que são descritos normalmente utilizando estruturas matemáticas contínuas, os modelos do universo virtual são discretos (porque todas as informações contidas nesse mundo são codificadas em seqüências finitas de zeros e uns). Assim, as estruturas matemáticas utilizadas na descrição dos fenômenos que ocorrem nesses dois tipos de universos são diferentes. Em Informática, a área que estuda diferentes maneiras de descrever modelos matemáticos do mundo virtual se chama *semântica formal*.

Inicialmente, foram estudados modelos para descrever sistemas seqüenciais. A medida em que novos conceitos foram surgindo, por exemplo, concorrência, orientação a objetos, mobilidade, etc., os modelos semânticos vão sendo adequados a estas novas realidades. Portanto, o desenvolvedor de uma nova linguagem e/ou paradigma de computação deve se preocupar em descrever o modelo matemático correspondente para que estes conceitos possam ser bem entendidos e analisados.

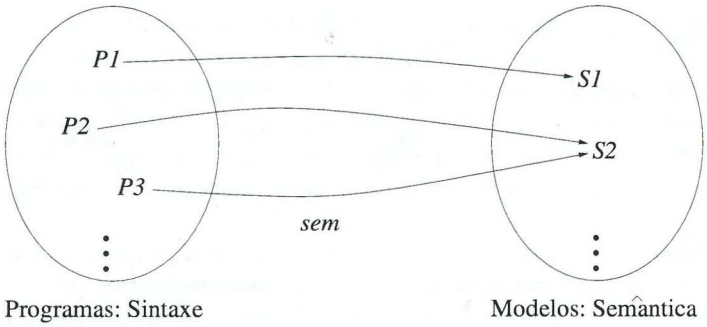

Figura 1: Mapeamento de Sintaxe para Semântica

1.2 Especificação Formal

Para desenvolver um sistema de software, não seria viável exigir que cada projetista possua conhecimentos profundos de como construir modelos semânticos. Mas, como ele pode então construir o modelo matemático do seu sistema? A resposta é: da mesma forma em que ele não precisa conhecer a linguagem de máquina do computador que está usando para escrever um programa que execute neste computador. Ou seja, podemos colocar ao seu dispor uma linguagem que permita que o sistema seja descrito de forma abstrata, completa e precisa, e para a qual exista uma maneira automática de se obter um modelo matemático (semântico). Estas linguagens são chamadas linguagens formais (pois tem sintaxe precisamente definida) e podem ser utilizadas em vários níveis de abstração, por exemplo, linguagens de programação, linguagens de especificação, etc. A Figura 1 mostra um exemplo de um mapeamento que associa a cada programa um modelo matemático que representa seu comportamento (semântica).

1.3 Verificação Formal

O modelo matemático de um sistema produzido por sua semântica pode ser analisado rigorosamente de diversas formas através das técnicas de verificação formal. Pode-se verificar se este modelo apresenta propriedades essenciais ou importantes para o sistema como, por exemplo, que o sistema não "trava" (ausência de dead-locks), que o sistema atende a toda requisição que lhe é feita ou que toda a operação iniciada chega ao fim (terminação). De modo mais completo, pode-se verificar se o sistema é uma implementação correta de uma especificação formal, ou seja, comporta-se exatamente conforme esperado (Figura 2). Usualmente, esta primeira abordagem de verificação formal é chamada de verificação de propriedades e a segunda é chamada de verificação de correção de implementação. Devido à natureza matemática das técnicas de verificação formal, elas produzem afirmações gerais válidas para todos os

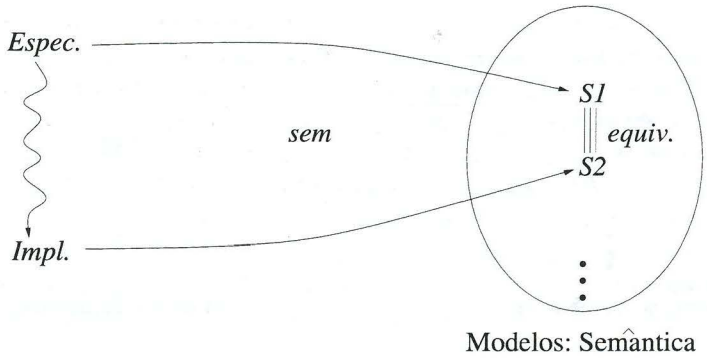

Figura 2: Correção de uma Implementação

comportamentos do sistema, mesmo quando o número de casos possíveis é infinito. Esta é a principal vantagem da verificação formal sobre os métodos usuais de teste e validação de sistemas. Deve-se observar que a verificação formal só pode ser realizada porque tanto o sistema como suas propriedades e especificação são apresentadas em linguagens com uma sintaxe e semântica formalmente definidas.

A seguir, os tópicos de especificação, semântica e verificação formal serão discutidos e exemplificados nas seções 2, 3 e 4, respectivamente.

2. Especificação Formal

Para podermos fazer uma verificação formal (matemática) de propriedades de um sistema, o primeiro passo é construir um modelo matemático que descreve o comportamento deste sistema. Para a construção destes modelos são utilizadas linguagens/métodos de *especificação formal.* Uma especificação é uma descrição de alto-nível (abstrata) do sistema a ser construído. Especificações devem ser compactas, precisas e não-ambíguas. Uma especificação formal é uma descrição de um sistema feita em uma linguagem com sintaxe e semântica precisamente definidas, ou seja, definidas utilizando-se conceitos matemáticos.

Alguns conceitos matemáticos são particularmente importantes para a especificação de sistemas: teoria dos conjuntos, lógica, álgebra abstrata, teoria das categorias, teoria dos domínios, etc. Com estas ferramentas é possível construir sistemas formais [Mar88], e estudar certas propriedades desses sistemas, como consistência, coerência e completeza.

Inicialmente, os métodos de especificação formal que surgiram se apoiavam na construção de uma *máquina abstrata* que representava o comportamento do sistema. Deste modo, a semântica (comportamento) de um sistema era dada pela definição

desta máquina abstrata com estados discretos e por seqüências de operações computacionais que iam modificando o estado da máquina. Exemplos desta abordagem, chamada operacional, são: máquinas de estados finitos e VDL (Vienna Definition Language [Pag81, BBFM82]). Esta abordagem é bastante voltada para a implementação, e por isso seu uso foi sendo substituída por abordagens onde fosse possível abstrair dos detalhes de implementação, e só se preocupar com o comportamento do sistema.

As abordagens denotacional e axiomática surgiram quase simultaneamente, cada uma baseada em um formalismo matemático. A abordagem denotacional está baseada na teoria dos domínios de Scott [Sto77]. Nesta abordagem, é construído um modelo do sistema utilizando estruturas matemáticas conhecidas (como conjuntos, funções, etc.). Depois de mostrar que o modelo construído é equivalente a realidade (não existe uma prova formal para esta equivalência, o projetista deve confiar que sua especificação modela adequadamente a realidade), são construídas as operações que se quer realizar no sistema. Estas operações são normalmente modeladas por funções que descrevem como a operação modifica o estado do sistema, ou seja, a semântica de cada operação é descrita pela relação entre os estados do sistema antes e depois da execução da operação (e não por uma seqüência contendo todos os estados pelos quais o sistema passa durante a execução da operação, como na abordagem operacional). Exemplos de métodos baseados nesta abordagem são: VDM (Vienna Development Method [BJ78]), Z [Abr85].

A abordagem axiomática se baseia na álgebra abstrata e na lógica formal. Esta abordagem surgiu das pesquisas sobre métodos de verificação de programas, mais especificamente sobre como provar que um programa possui certas propriedades [LB86, Man74]. Esta verificação é baseada em asserções sobre os valores de entrada e saída de um trecho de programa. Dijkstra desenvolveu a idéia de, ao invés de tentar verificar a correção de um programa utilizando axiomas, seria possível assegurar a correção derivando o programa de uma especificação das propriedades que ele deve apresentar, ou seja, utilizar um método axiomático para fazer a síntese de um programa, e não apenas uma análise. Assim surgiu a abordagem axiomática para especificação de programas. Nesta abordagem, o modelo do sistema é dado por um conjunto de asserções que ele deve satisfazer (propriedades do sistema).

Essa abordagem deu origem a uma linha de linguagens de especificação algébricas [GTW77, GT86, GHW85], linha que até o presente não conseguiu grande penetração em ambientes não acadêmicos. Uma das razões dessa dificuldade é o tipo de raciocínio necessário para a definição dessas especificações que é próximo ao utilizado no paradigma de programação funcional e menos popular que o raciocínio imperativo, baseado nas noções de algoritmos e estados. Uma nova geração de linguagens algébricas está no entanto sendo desenvolvida [CoF99, SRI98, DF98], e espera-se que a maior maturidade alcançada na área e seus ambientes de desenvolvimento mais completos, com o apoio de ferramentas de apoio à especificação e verificação de programas,

permitam que alcancem maior sucesso.

Com o aparecimento de CCS (Calculus of Communicating Systems) [Mil80], cálculo algébrico destinado a descrever a analisar o comportamento de sistemas paralelos, a abordagem operacional voltou a ser utilizada. Nos últimos anos, surgiram outros métodos operacionais com um propósito específico: especificar sistemas concorrentes e distribuídos. Nestes métodos, os sistemas são descritos, geralmente, como um conjunto de processos que evoluem independentemente uns dos outros. O comportamento do sistema é dado pelas seqüências de eventos possíveis de ocorrer no sistema. Com esses métodos, pode-se provar propriedades como ausência de deadlocks e a terminação de um processo. CSP [Hoa85a] e LOTOS [BB87] são outros exemplos de métodos que seguem esta abordagem.

Na prática, os métodos que seguem a abordagem operacional são tipicamente mais fáceis de compreender por não-especialistas do que os das outras abordagens. Mas muitas vezes o nível de detalhe é muito grande tornando as especificações complexas. Os métodos denotacionais oferecem um nível maior de abstração em comparação com os operacionais, mas muitas vezes a notação extremamente matemática dificulta seu entendimento. Na abordagem axiomática, um sistema é descrito de maneira mais abstrata do que nas outras, pois não é dado um modelo concreto para o sistema. A idéia é que qualquer modelo concreto que satisfaça os axiomas pode ser considerado como um modelo para o sistema, o que dá uma boa flexibilidade para implementações. Porém, não-matemáticos tendem a ter dificuldades em visualizar um sistema especificado através de axiomas.

Como cada abordagem possui pontos fracos e fortes, surgiram também métodos que combinam mais de uma abordagem. Gramáticas de grafos [Ehr79, EHK+97] é um exemplo disso. Neste formalismo, os estados de um sistema são descritos por estruturas algébricas (grafos) e o comportamento do sistema é descrito de maneira operacional através de mudanças de estados. LOTOS, linguagem para a especificação de sistemas concorrentes, possui também construções para a especificação de dados sob o paradigma algébrico, e CASL, linguagem de especificação algébrica, possui extensão para o tratamento de concorrência. Finalmente, outros exemplos de abordagem mista unem linguagens desenvolvidas independentemente, como CSP-Z [Fis96].

2.1 Exemplo: Ferrovia

Nesta seção utilizaremos como exemplo a descrição de uma ferrovia bem simples. A ferrovia consiste de *trens* que recebem instruções de uma *central* indicando para onde eles devem se dirigir. A malha ferroviária terá dois tipos de componentes: *trilhos* normais e *desvios*, que são também controlados pela central.

O comportamento da ferrovia deve ser o seguinte:

1. Inicialmente, cada trem está em uma posição diferente da malha ferroviária, esperando por uma informação da central sobre o próximo trecho a ser percorrido.

e na central está sinalizado que ela deve enviar tais informações.

2. A central envia para cada trem a informação sobre o próximo trecho a ser percorrido.

3. Os trens então começam a se movimentar, e, quando chegarem na posição informada pela central, enviam um sinal para ela informando que já estão na posição final e esperam novas instruções.

4. Para evitar desastres, o trens têm sensores que indicam se a próxima posição para a qual eles querem se movimentar está livre ou não. Se estiver ocupada, o trem fica esperando a posição ficar livre.

5. Quando o trem estiver em um desvio, e este estiver no momento conectado ao caminho que o trem quer seguir, ele segue. Senão, ele solicita a central que mude o desvio para o outro caminho e fica esperando (para simplificar, vamos assumir que cada desvio tem somente duas posições).

6. Quando a central receber uma solicitação de troca de caminho de desvio, esta só poderá ser efetuada se os trilhos imediatamente adjacentes ao desvio estiverem livres (o desvio em si estará ocupado, pois uma troca só acontece por solicitação de um trem que está no desvio).

Note que os trens e a central trabalham em paralelo: em um mesmo momento, podemos ter movimentação dos dois trens, e atividades na central. Até mesmo uma mesma entidade pode realizar mais de uma tarefa ao mesmo tempo: por exemplo, a central pode enviar mensagens aos dois trens ao mesmo tempo. Portanto, os passos enumerados acima não são seguidos seqüencialmente.

2.2 Gramáticas de Grafos

Técnicas de engenharia de software devem poder garantir que um um programa é realmente uma solução para um dado problema. Esta tarefa envolve a formalização do problema, inicialmente dado por informações e requisitos descritos de maneira informal, bem como a transformação desta formalização em código executável. Para formalizar um problema é desejável uma maneira natural e intuitiva de descrição. Para se executar provas formais de propriedades do sistema é necessário que a semântica da especificação seja descrita por modelos matemáticos. Propriedades essenciais de sistemas complexos, como distribuição, paralelismo, comunicação, interfaces gráficas sofisticadas, devem ser consideradas neste processo. O fato de serem formais e intuitivas ao mesmo tempo, e também de poderem tratar com simplicidade aspectos de concorrência e distribuição de sistemas, faz de gramáticas de grafos um método promissor para o desenvolvimento de software confiável.

Grafos são um meio natural para explicar situações complexas de modo intuitivo. Regras podem ser utilizadas para descrever aspectos dinâmicos de sistemas. Gramáticas de grafos são uma generalização de gramáticas de Chomsky, subtituindo strings por grafos. Existem várias abordagens diferentes para gramáticas de grafos. As principais abordagens são a algébrica [Ehr79, Löw93], a NCL [Roz87], a algorítmica [Nag87] e a delta [EKR91]. Neste curso nós seguiremos a abordagem algébrica de gramáticas de grafos. Nesta abordagem foram feitas muitas investigações na área de concorrência.

Pesquisas na área de gramáticas de grafos iniciaram nos anos 70, e métodos, técnicas e resultados nesta área já foram estudados e aplicados um uma grande variedade de campos da Informática, como teoria de linguagens formais, reconhecimento e geração de imagens, construção de compiladores, engenharia de software, modelagem de sistemas concorrentes e distribuídos, projeto e teoria de bancos de dados, etc. (veja por exemplo, [Nag92]). Gramáticas de grafos podem também ser consideradas como uma generalização de redes de Petri [Cor95, KR96], permitindo mudanças dinâmicas na topologia do sistema e referências entre tokens. Estas propriedades são bastante úteis na modelagem de comunidades de objetos que fazem referências e se comunicam entre si e que podem ser criados e destruí dos durante o ciclo de vida do sistema.

A descrição de gramáticas de grafos dada a seguir é dada de maneira informal e intuitiva. Para as definições formais de gramáticas de grafos algébricas veja [Ehr79, Löw93] ou [EHK+97, Kor96, Rib96].

Gramáticas de grafos generalizam gramáticas de Chomsky usando grafos ao invés de strings. Diferente de regras em gramáticas de Chomsky, uma regra de grafos $r : L \to R$ não consiste somente dos grafos L (lado esquerdo) e R (lado direito), mas também de uma parte adicional: um (homo)morfismo parcial de grafos r que mapeia vértices e arcos de L em vértices e arcos de R de maneira compatível. Compatibilidade aqui significa que cada vez que um arco e_L for mapeado em um arco e_R, então o vértice origem de e_L deve ser mapeado para o vértice origem de e_R e o mesmo para o vértice destino.

Exemplo 1 *Considere grafo mais a esquerda da Figura 3. Este grafo tem 3 vértices (* $\#_1$, $\#_2$ *e* $\blacksquare$*), e dois arcos (conectando o trem ao trilho 1, e conectando os dois trilhos. Um morfismo de grafos está representado pelas linhas tracejadas na Figura (a). As linhas tracejadas na Figura (b) não formam um morfismo de grafos porque os vértices de origem e destino do arco que conecta os trilhos não são preservados.*

Na abordagem que nós seguiremos, uma gramática de grafos especifica um sistema em termos de estados – modelados por *grafos* (ou estruturas parecidas com grafos) – e mudanças de estados – modeladas por *derivações*. A seguinte interpretação operacional de uma regra $r : L \to R$ descreve a base desta abordagem de especificação:

- Ítens em L que não tem uma imagem em R são *removidos*.

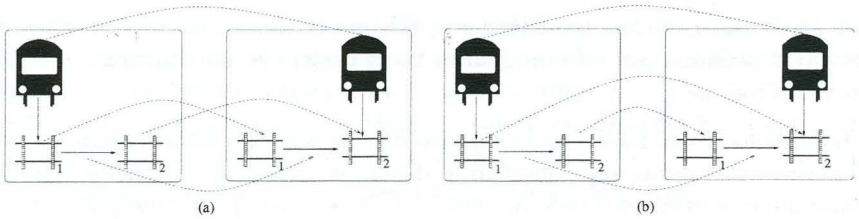

Figura 3: (a) Morfismo de grafos (b) Não é morfismo de grafos

- Ítens em L que são mapeados para R são *preservados*.
- Ítens em R que não tem uma pré-imagem em L são *criados*.

Ao invés de usar grafos simples que somente consistem de arcos e vértices, nós geralmente usamos algum mecanismo de tipagem nos grafos. Neste capítulo usaremos grafos com rótulos, ou seja, cada arco/vértice será rotulado com um elemento de um alfabeto Σ de rótulos. No exemplo da Figura 3, um conjunto de rótulos é usado para distinguir os vértices que são usados: existem vértices do tipo trem e do tipo trilho. O uso de mecanismos de tipagem faz as especificações se tornarem mais simples, compactas e fáceis de entender. Outro mecanismo de tipagem para grafos é o uso de atributos [LKW93] (onde tipos abstratos de dados são usados) e grafos-tipo [Kor94a, CMR96] (onde um grafo é utilizado como tipo). Na seção 2.2.1 utilizaremos atributos. A abordagem algébrica para gramáticas de grafos é definida utilizando-se conceitos de teoria das categorias [BW90]. A utilização desta ferramenta matemática permitiu a construção de uma abordagem paramétrica em relação ao tipo de grafos utilizados. Isto dá ao usuário uma grande flexibilidade para escolher as estruturas que mais se adequam a modelagem de seu sistema em particular.

Uma *gramática de grafos* consiste dos seguintes componentes:
um conjunto de rótulos,
conjuntos de atributos,
um grafo inicial e
regras.

2.2.1 Ferrovia Especificada Usando Gramáticas de Grafos

Para especificar a ferrovia utilizando gramáticas de grafos, o primeiro passo é descrever os possíveis estados da ferrovia usando grafos:

- Cada símbolo ⌗ modela um trecho da ferrovia. O identificador deste trecho é dado pelo número no interior do trilho. Se este número for preto, o trecho está livre, caso o número for branco, já está ocupado por um trem.

- Os arcos entre trechos modelam que existem conexões entre os trechos da ferrovia, indicando o sentido no qual os trens podem se movimentar.

- Os símbolos  modelam os desvios da ferrovia. Os estado atual do desvio é descrito pelos arcos que conectam o desvio a trilhos: arcos cheios significam o caminho que está conectado ao desvio no momento, arcos tracejados indicam o caminho do desvio que não está conectado no momento.

- Os símbolos  modelam os trens. Um arco modela o posicionamento corrente do trem na ferrovia e o número dentro do trem é o próximo trecho onde o trem deve ir.

- O símbolo 

é a central que controla os trens.

Para a tipagem dos elementos dos grafos que representam um estado, utilizaremos os seguintes conjuntos:

Conjunto de rótulos: Os vértices serão rotulados por elementos do conjunto $\Sigma =$ {  ,  ,

}

Atributos: Cada vértice poderá ter como atributo um elemento do conjunto $\{1, \ldots, 10,$, ..., $\}$. Para os trens utilizaremos apenas os números pretos, para os trilhos os pretos e os brancos, e para a central não serão utilizados atributos.

Figura 4: Ferrovia

A Figura 4 descreve um estado inicial possível para uma ferrovia, onde a malha ferroviária é composta por 10 posições (2 desvios e 8 trilhos simples), dois trens e uma central. No momento, o trem posicionado no trilho 2 quer se movimentar para o trilho 3, e o trem posicionado no trilho 6 quer se movimentar para o trilho 5. O desvio 3 está conectado ao trilho 4, e o desvio 8 ao trilho 10.

O funcionamento da ferrovia será modelado através de regras:

Funcionamento dos Trens: Figura 5

Regra *move*: O trem está em um trilho n que tem como vizinho um trilho m livre (número preto), e o trem quer se movimentar para a posição m. Nesta situação, o movimento pode ocorrer, ficando no final o número m preto, o número n branco, e o trem na posição m. A cada movimentação, o trem envia para a central uma mensagem indicando que está pronto para receber novas instruções sobre o seu destino.

Regra *entraDesvio*: O trem está em um trilho n que tem como vizinho um desvio m livre (número preto), e o trem quer se movimentar para a posição m. O comportamento nesta situação é análogo ao da regra *move*.

Regra *saiDesvio*: O trem está em um desvio n que atualmente está conectado ao trilho m, este trilho está livre (número preto), e o trem quer se movimentar para a posição m. O comportamento nesta situação é análogo ao da regra *move*.

Regra *pedeDesvio*: O trem está em um desvio n que atualmente não está conectado ao trilho m, para o qual o trem quer se movimentar. Neste caso, o trem permanece na posição atual, e envia uma mensagem para a central para o desvio ser mudado.

Funcionamento da Central: Figura 6

Regra *destTT*: A central, ao receber uma mensagem solicitando o próximo trecho de um trem posicionado em um trilho m que tem como vizinho um trilho n, muda o destino do trem para n.

Regra *destDT* A central, ao receber uma mensagem solicitando o próximo trecho de um trem posicionado em um desvio m que está conectado a um trilho n, muda o destino do trem para n.

Regra *destTD* (Não mostrada na Figura) Análoga às regras acima, permitindo que um trem entre em um desvio.

Regra *desvia*: A central recebe uma mensagem para mudar um desvio de m2 para m1. Se os caminhos conectados ao desvio estiverem livres, a central modifica o desvio.

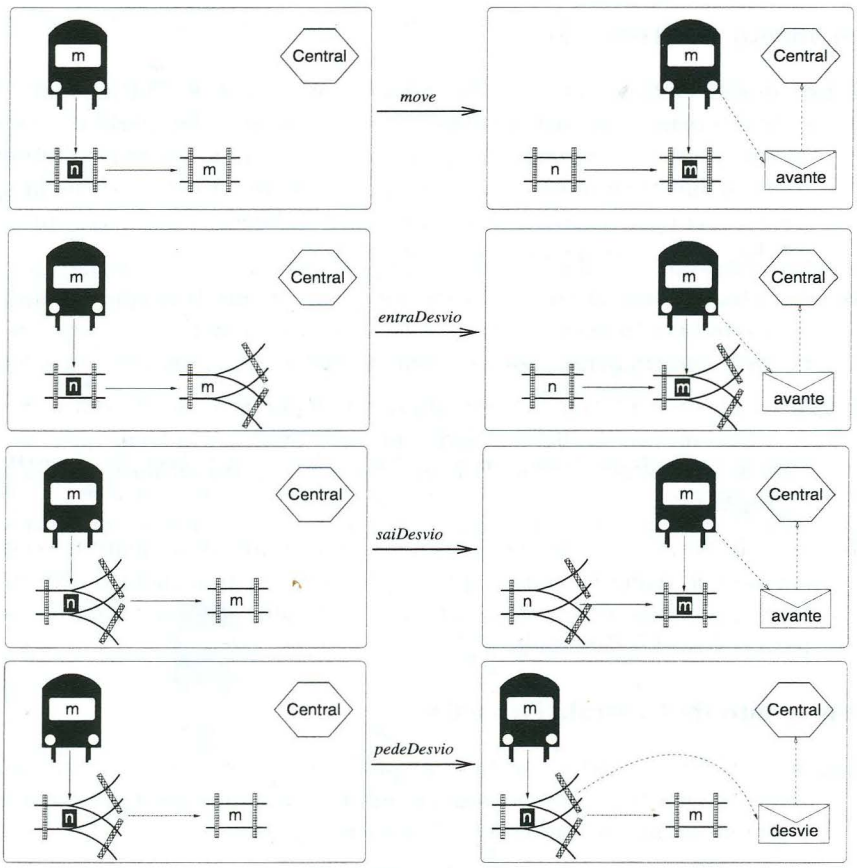

Figura 5: Regras dos Trens

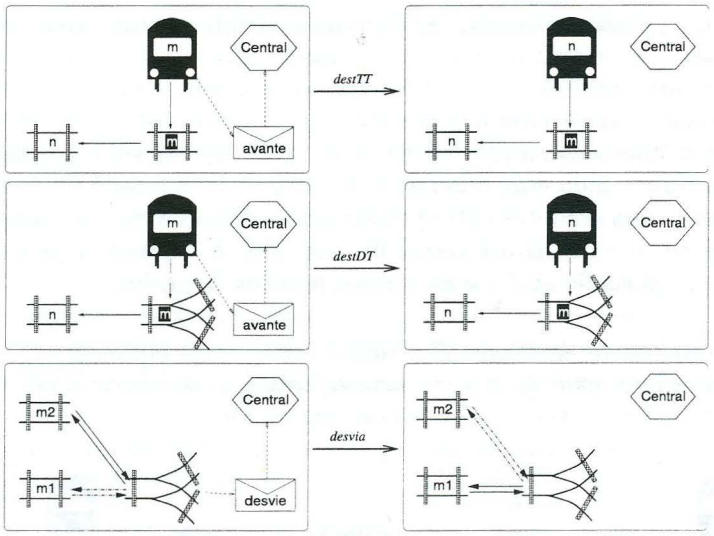

Figura 6: Regras da Central

2.2.2 Comportamento de uma Gramática de Grafos

O comportamento de um sistema modelado por uma gramática de grafos pode ser descrito pelas aplicações das regras da gramática a grafos que representam os estados do sistema. A aplicação de uma regra a um grafo IN, chamada de *passo de derivação*, é possível se existe uma ocorrência do lado esquerdo da regra no grafo IN. Esta ocorrência (chamada *match* em inglês), é modelada por um morfismo total de grafos, o que intuitivamente significa que todos os elementos do lado esquerdo da regra devem estar presentes em IN para que a regra possa ser aplicada. O resultado da aplicação de uma regra $r : L \rightarrow R$ a um grafo IN é obtido através dos seguintes passos:

1. Adicionar a IN tudo que for criado pela regra (ítens que fazem parte do lado direto R da regra, mas não do lado esquerdo L).

2. Remover do grafo resultante do passo 1 tudo que deve ser removido pela regra (ítens que fazem parte do lado esquerdo L e não do lado direito R).

3. Remover arcos pendentes. Este passo é necessário no caso de haverem arcos conectados a vértices removidos no passo 2. Como o resultado da aplicação de uma regra deve ser um grafo, este arcos devem ser removidos também. Intuitivamente, podemos comparar este fenômeno com a desalocação de uma variável

para a qual existem pointers: neste caso os pointers seriam destruídos automaticamente.

Apesar da ordem natural para a construção do resultado parecer ser passos 2, 3 e 1, escolhemos a ordenação acima por ela ser mais geral: funcionará corretamente mesmo para os casos onde o lado esquerdo não é injetivamente mapeado no grafo IN (neste caso, podem ocorrem conflitos entre preservação e remoção de um vértice).

Formalmente, o conceito de passo de derivação é capturado pela definição de PushOut na categoria de grafos e morfismos parciais de grafos.

Exemplo 2 *Aplicação de Regra. Na Figura 7 IN é transformado em OUT usando a regra entraDesvio aplicada à ocorrência referente ao mapeamento de* n *para* 2 *e* m *para* 3.

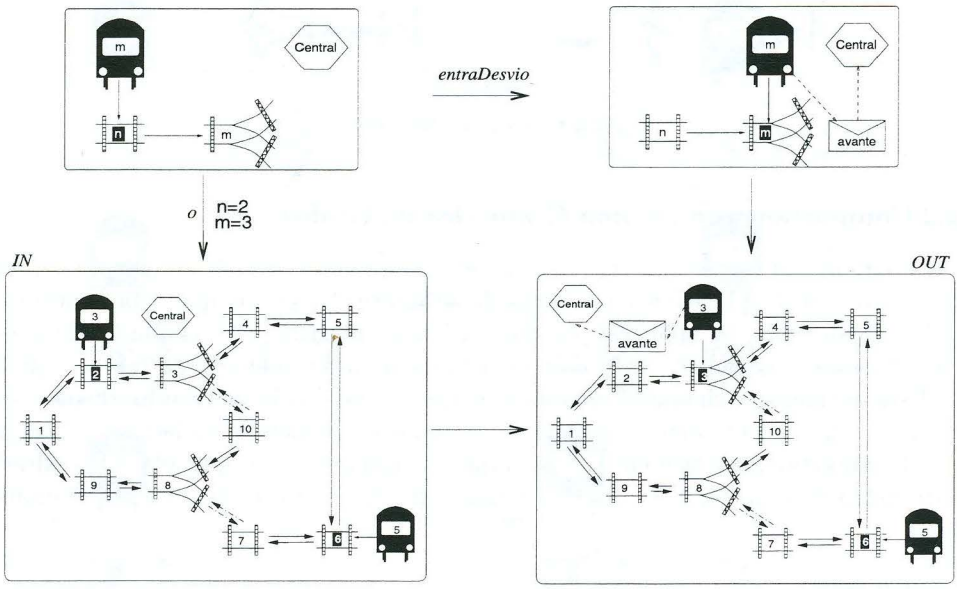

Figura 7: Passo de Derivação

Em uma gramática de grafos, vários passos de derivação podem ocorrer ao mesmo tempo. Por exemplo, na situação da Figura 4, o trem no trilho 2 pode se movimentar para o desvio 3 (usando a regra *entraDesvio* enquanto o trem no trilho 6 se movimenta para o trilho 5 (usando a regra *move*). Além disso, quando existem várias regras que podem ser aplicadas que tentam remover o mesmo ítem, uma delas é escolhida não-deterministicamente. Por exemplo, quando a central recebe uma solicitação de novo

destino de um trem que está num trilho que tem como vizinhos um trilho n e um desvio m, ela deve escolher entre enviar para o trem o destino n (usando a regra $destTT$) ou o destino m (usando a regra $destTD$). Esta escolha ocorre de maneira aleatória.

Como podemos saber se dois trens nunca irão colidir nesta ferrovia? Imagine que temos a situação da Figura 8 (que mostra apenas uma parte da ferrovia). A regra que poderia ser aplicada nesta situação é a regra *move*. Como os dois trens estão tentando modificar o atributo do trilho 5 (ou seja, remover o 5 branco e colocar um preto no lugar), não podemos aplicar a regra *move* para os dois trens ao mesmo tempo, apenas um conseguirá ir para a posição 5. Agora, imagine que o trem da posição 6 conseguiu se movimentar para a 5, e agora quer ir para a 4, e o outro continua na posição 4, querendo ir para a 5. Nesta situação, nenhuma regra poderá ser aplicada, pois a aplicação da regra *move* exige que a posição destino do trem esteja livre. Então, a ferrovia pararia. Caso se queira tratar este tipo de situação, novas regras deveriam ser incluídas na especificação.

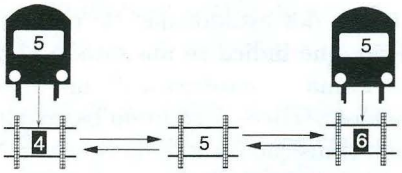

Figura 8: Parte de um Estado

3. Semântica Formal

Uma linguagem deve permitir a expressão de programas/especificações de maneira rica e flexível. Isto implica em que, usualmente, existem vários programas/especificações que produzem o mesmo comportamento. A escolha de um modelo semântico para uma linguagem deve ser baseada na nas características que são relevantes para serem analisadas e para distinguir programas. Existem muitos tipos de modelos semânticos que podem ser construídos para definir a semântica de uma linguagem. Um dos mais simples é o conjunto de estados gerados pelo sistema durante sua execução. Neste caso, o significado de um programa/especificação X estaria definido por um conjunto de estados $est(X)$. Duas especificações A e B seriam consideradas equivalentes se os conjuntos de estados $est(A)$ e $est(B)$ fossem iguais. Este tipo de semântica é bastante simples (um conjunto), fácil de analisar, e extremamente abstrata, já que as informações sobre como cada estado foi obtido não faz parte da semântica. Mas, se A e B forem sistemas concorrentes, e quisermos saber se A realiza suas tarefas utilizando mais paralelismo do que B? Neste caso, as informações contidas nesta semântica não

seriam suficientes, pois não há como chegar a conclusões sobre ações que ocorreram que paralelo olhando apenas o conjunto de estados gerados pelas especificações. Portanto, dependendo dos aspectos relevantes na linguagem, um modelo semântico pode ser mais adequado que outros.

Como discutido acima, para permitir a análise de concorrência em um modelo semântico, este modelo deve incluir as ações que podem ser realizadas no sistema. Na seção 3.1 será mostrado um tipo de modelo semântico possível para expressar concorrência: sistemas de transição, e também como obter o sistema de transição que define o significado de cada gramática de grafos. Na seção 3.2 discutiremos brevemente alguns outros modelos adequados para sistemas concorrentes e mostraremos uma classificação destes modelos.

3.1 Semântica de Gramáticas de Grafos usando Sistemas de Transição

Um sistemas de transição é um modelo matemático bastante simples e ao mesmo tempo expressivo para descrever o comportamento de um sistema. A idéia básica é que o sistema é descrito através dos estados que ele pode atingir e de uma relação, chamada relação de transição, que indica se um estado $s1$ pode ser obtido a partir de outro estado $s0$. Desta forma, a semântica de um sistema é representada por uma estrutura de grafo, onde os vértices descrevem os estados e os arcos as possíveis transições. Usualmente, cada transição é rotulada com uma informação sobre a ação que, quando realizada, permitiu a mudança de estados descrita pela transição.

Uma *sistema de transição* consiste dos seguintes componentes:
um conjunto de estados S,
um conjunto de rótulos para transições L,
um estado inicial $s0 \in S$ e
uma relação de transição $\rightarrow \subseteq S \times L \times S$.

Para construir o sistema de transição que descreve o comportamento de uma gramática de grafos, utilizaremos o conceito de passo de derivação (definido na seção 2.2.1), que define quando se pode atingir um estado (grafo) $s1$ a partir de um estado $s0$: isto só é possível se existe uma regra r da gramática que, quando aplicada a $s0$ (usando uma ocorrência m), gera o estado $s1$, ou seja, $s0 \stackrel{r,m}{\Longrightarrow} s1$. Porém, só consideraremos estados que podem ser atingidos a partir do grafo inicial da gramática. Por isso, utilizaremos o conceito de derivação seqüencial: uma derivação seqüencial de uma gramática GG é uma seqüência $s0 \stackrel{r1,m1}{\Longrightarrow} s1 \stackrel{r2,m2}{\Longrightarrow} s2 \stackrel{r,m}{\Longrightarrow} \ldots$ onde cada regra ri pertence a gramática e $s0$ é o grafo inicial da gramática. Uma derivação seqüencial pode ser infinita ou finita, terminando em um grafo sn.

Para uma gramática de grafos composta pelo conjunto de rótulos Rot, pelos atributos Atr, pelo grafo inicial Ini e pelo conjunto de regras R, podemos obter um

sistema de transição que descreve seu comportamento da seguinte forma [†]:

Conjunto de estados: Como em uma gramática de grafos os estados são modelados por grafos, o conjunto de estados pode ser definido como um conjunto contendo todos os grafos que podem ser gerados utilizando os rótulos de Rot e atributos de Atr a partir do grafo inicial Ini utilizando regras de R, ou seja, o conjunto de estados s tais que $s = Ini$ ou existe uma derivação seqüencial $Ini = s0 \overset{r1,m1}{\Longrightarrow} s1 \overset{r2,m2}{\Longrightarrow} s2 \overset{r3,m3}{\Longrightarrow} \ldots \overset{r,n}{\Longrightarrow} s$.

Conjunto de rótulos: As transições serão rotuladas com o passo de derivação ao qual ela corresponde.

Estado inicial: O estado inicial do sistema de transição é o grafo inicial da gramática.

Relação de transição: Entre dois estados $s0$ e $s1$ pode existir uma transição rotulada por d caso exista um passo de derivação $d = s0 \overset{r,o}{\Longrightarrow} s1$, onde r é uma regra da gramática considerada.

Exemplo 3 *O comportamento da gramática de grafos que modela o sistema de trens descrita na seção 2.2.1 pode ser modelado pelo sistema de transição mostrado (parcialmente) na Figura 9. Para descrever os estados e as transições, utilizaremos a convenção que o trem inicialmente na posição* 2 *será chamado de* A *e o na posição* 6 *de* B. *Para descrever cada estado, diremos apenas a posição e o destino dos trens e as mensagens a serem processadas pela central, o resto do grafo continua como na Figura 4.*

Os estados $s0$ *a* $s10$ *representam os seguintes grafos:*

$s0$: *grafo inicial* IN *(Figura 4), ou seja,* $A^2 \mapsto 3$, $B^6 \mapsto 5$, $MSG = \emptyset$ *(trem* A *na posição* 2 *tem como destino a posição* 3, *trem* B *na posição* 6 *tem como destino a posição* 5, *e não há mensagens na central);*

$s1$: $A^3 \mapsto 3$, $B^6 \mapsto 5$, $MSG = \{\mathsf{avante(A)}\}$;

$s2$: $A^2 \mapsto 3$, $B^5 \mapsto 5$, $MSG = \{\mathsf{avante(B)}\}$;

$s3$: $A^3 \mapsto 2$, $B^5 \mapsto 6$, $MSG = \{\}$;

[†]Na abordagem algébrica para gramáticas de grafos, devido ao uso da construção categórica pushout para definir os passos de derivação, resultado da aplicação de uma regra a uma ocorrência é, na realidade, uma classe de grafos isomórficos. Portanto, a construção do sistema de transição deveria considerar como estados classes de grafos e como transições classes de passos de derivação. Porém, deve-se observar que este tratamento considerando classes isomórficas como objetos traz problemas na definição de noções adequadas de composição de transições (para mais detalhes, veja [CMR+97].

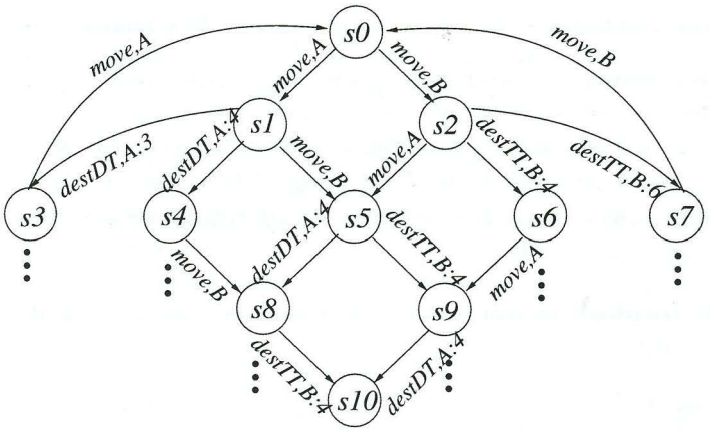

Figura 9: Sistema de Transição

$s4$: $A^3 \mapsto 4$, $B^5 \mapsto 6$, $MSG = \{\}$;

$s5$: $A^3 \mapsto 3$, $B^5 \mapsto 5$, $MSG = \{\text{avante(A), avante(B)}\}$;

$s6$: $A^2 \mapsto 3$, $B^5 \mapsto 4$, $MSG = \{\}$;

$s7$: $A^2 \mapsto 3$, $B^5 \mapsto 6$, $MSG = \{\}$;

$s8$: $A^3 \mapsto 4$, $B^5 \mapsto 5$, $MSG = \{\text{avante(B)}\}$;

$s9$: $A^3 \mapsto 3$, $B^5 \mapsto 4$, $MSG = \{\text{avante(A)}\}$;

$s10$: $A^3 \mapsto 4$, $B^5 \mapsto 4$, $MSG = \{\}$.

As transições representam as mudanças de estado possíveis, e são rotuladas pelos correspondentes passos de derivação. Na Figura 9, utilizamos a seguinte convenção para os rótulos: $r, T : d$, onde r é o nome da regra utilizada, T é o trem envolvido no passo de derivação, e d é o novo destino do trem T (caso a regra não envolva mudança de destino, a parte $: d$ não é utilizada). Por exemplo, no estado $s0$ podemos utilizar a regra move para movimentar o trem A, indo então para o estado $s1$ (no qual o trem A está na posição 3), ou utilizar a regra move para movimentar o trem B, indo para o estado $s2$.

Um sistema de transição descreve um sistema como um todo, onde cada estado pelo qual o sistema pode passar é representado explicitamente. Por isso, ele é considerado um *modelo de sistema*. Em um sistema de transição, podemos identificar as ações que podem ocorrer em paralelo do seguinte modo: se existem transições $s0 \xrightarrow{r1} s1 \xrightarrow{r2} s2$ e

$s0 \xrightarrow{r2} s1' \xrightarrow{r1} s2$, então as ações rotuladas com $r1$ e $r2$ poderiam ocorrer em paralelo. No exemplo acima, as ações $move, A$ e $move, B$, a partir do estado $s0$ poderiam ocorrem em paralelo, gerando em um passo o estado $s5$. Este tipo de descrição de paralelismo se chama abordagem *interleaving* (ou *entrelaçamento*). Em cada estado, estão descritas explicitamente todas as escolhas possíveis de próximo estado, através das transições que saem deste estado. Por exemplo, do estado $s1$ o sistema pode evoluir para os estados $s3$, $s4$ ou $s5$. Um modelo deste tipo se chama *branching structure* (*estrutura de bifurcações*). Note que, no exemplo, uma escolha entre seguir para $s3$ e $s4$ representa realmente uma escolha entre dois caminhos possíveis para o sistema (ou seja, as ações que levam a $s3$ e $s4$ estão em conflito). Já uma escolha entre seguir para $s4$ ou $s5$ é simplesmente uma escolha da ordem na qual as ações $move, B$ e $destDT, A : 4$ são executadas, já que estas ações são independentes entre si.

Na próxima seção discutiremos outras abordagens para descrever sistemas concorrentes.

3.2 Modelos Semânticos para Sistemas Concorrentes

Os modelos para a descrição de sistemas concorrentes podem ser classificados segundo três dimensões [SNW93]:

Modelos de *Sistemas* ou *Comportamento*: Em um modelo de sistema, os estados pelos quais o sistema pode passar são representados explicitamente, e cada estado ocorre apenas uma vez no modelo, ou seja, os modelos usualmente contém ciclos. Em modelos de comportamento, considera-se que nunca se pode voltar a um estado anterior, ou seja, uma vez ocorrida uma ação em um estado, este evento nunca mais poderá ser repetido. O que podemos fazer é executar uma ação análoga, mas que não será mais a ação original. Desta forma, não existem ciclos em modelos de comportamento.

Modelos *Interleaving* ou *True concurrency*: Em um modelo interleaving, a concorrência é modelada como não-determinismo, ou seja, se podemos escolher entre executar a e depois b ou b e depois a, então a e b podem também executar em paralelo. Em um modelo true concurrency (ou concorrência verdadeira), a concorrência é descrita através da independência entre as ações: se sabemos que a e b são independentes, então elas podem executar em paralelo. Portanto, em modelos true concurrency, ao invés de descrevermos as escolhas possíveis, descrevemos um sistema em termos de relações entre ações que permitam analisar a independência entre elas (usualmente, são utilizadas relações de causa e conflito). Alguns modelos true concurrency são capazes de expressar um maior número de situações onde paralelismo é possível do que modelos interleaving [KR98].

Modelos *Linear* ou *Branching Structure*: Em um modelo branching structure (*estrutura de bifurcações*), os pontos de escolha de um sistema são representados explicitamente. Em um modelo linear structure (*estrutura linear*), estes pontos não são representados explicitamente. Tipicamente, um modelo branching structure modela um sistema como um todo, e um modelo linear structure modela um sistema como um conjunto de computações (existe uma computação para cada conjunto de escolhas possível).

A seguir, descreveremos brevemente um exemplo de modelo matemático que se enquadra em cada combinação de dimensões:

Comp/Inter/Linear: Linguagens de Hoare [Hoa85b]
Neste modelo, o comportamento de um sistema é dado por um conjunto de palavras, onde cada palavra representa uma seqüência de ações possível do sistema. Por exemplo, a palavra *abaa* representa o fato de que podemos realizar a ação chamada de a seguida da ação b seguida de duas novas ocorrências de ações chamadas de a. Este modelo é de estrutura linear porque, considerando apenas uma palavra, não conseguimos verificar quais foram os pontos onde ocorreram escolhas não-determinísticas. Para isso, temos que comparar com outras palavras.

Comp/Inter/Branch: Árvores de Sincronização [Win85]
Uma árvore de sincronização é um sistema de transição acíclico onde todos os estados são alcançáveis. Podemos obter uma árvore de sincronização a partir de um sistema de transição se desdobrarmos os ciclo, gerando novos estados. Por exemplo, se existe um ciclo $s0 \rightarrow s1$ e $s1 \rightarrow s0$, em uma árvore de sincronização este ciclo seria substituído por $s0 \rightarrow s1 \rightarrow s0' \rightarrow s1' \rightarrow s0''' \rightarrow s1'' \rightarrow cdots$.

Comp/TrueConc/Linear: Derivações Concorrentes [Kor94b]
A partir de uma derivação seqüencial de uma gramática de grafos, podemos obter uma derivação concorrente através de uma colagem dos grafos intermediários. Desta forma, perde-se a informação sobre a ordem na qual os passos de derivação ocorreram na derivação. Porém, através de análise pode-se inferir quais passos dependem de outros (relação de causalidade). Para cada possível escolhas não-determinísticas em um sistema teremos uma nova derivação concorrente. Assim, neste modelo semântico o comportamento de uma gramática é dado por um conjunto de derivações concorrentes (computações concorrentes). Este modelo é de comportamento porque um estado nunca pode ser repetido, podemos chegar a um estado isomórfico, mas ele não será o mesmo que um anterior.

Comp/TrueConc/Branch: Estruturas de Eventos [Win89]
Uma estrutura de eventos descreve o comportamento de um sistema através do

conjunto de ações que podem ser realizadas no sistema, chamadas de eventos, e de duas relações entre esses eventos: relação de causalidade (que diz quando um evento é a causa de um outro) e relação de conflito (que indica quando a ocorrência de um evento exclui a possibilidade de o outro ocorrer).

Sys/Inter/Linear: Sistemas de Transição Determinísticos
Pode-se descrever o comportamento de um sistema como um conjunto de sistemas de transição determinísticos, ou seja, onde não existem pontos de escolha (mas podem haver ciclos). Cada sistema de transição determinístico representa uma possível computação do sistema.

Sys/Inter/Branch: Sistemas de Transição
(Este modelo já foi explicado na seção anterior.)

Sys/TrueConc/Linear: Sistemas de Transição Determinísticos com Independência [Sta89]
Em um sistema de transição com independência, é adicionado ao sistema de transição uma relação de independência de transições, indicando quando duas transições são independentes entre si (e, portanto, podem executar em paralelo). Desta forma, as informações sobre a concorrência do sistema não seriam mais derivadas do não-determinismo, e sim da relação de independência. Pode-se descrever a semântica de um sistema como um conjunto de sistemas de transição determinísticos com independência.

Sys/TrueConc/Branch: Sistemas de Transição com Independência
Similar ao modelo acima, com a diferença que o sistema é descrito como um todo (escolhas não-determinísticas são representadas explicitamente), ou seja, a semântica é dada por um único sistema de transição.

4. Verificação Formal

No ciclo usual de desenvolvimento de software ou hardware, o método mais empregado para assegurar que o sistema funciona como desejado é o *teste* do sistema. Por este método, o programa ou modelo do sistema é executado para um conjunto de valores de entrada e observa-se o resultado obtido. Caso o resultado não seja o esperado, corrige-se o sistema. Caso o resultado seja aceitável, repete-se o teste com novos valores ou considera-se que o sistema está correto.

Na maioria dos sistemas reais, o número de casos que devem ser testados para cobrir todo o comportamento do sistema é muito grande ou infinito. Então o teste é feito por amostragem, testando-se o sistema em apenas algumas situações. Neste caso, há grande possibilidade de que erros existentes no sistema não sejam detectados.

A verificação formal consiste em utilizar técnicas matemáticas para assegurar que um sistema computacional apresenta uma certa propriedade ou satisfaz sua especi-

ficação. O uso de técnicas matemáticas permite considerar todos os casos possíveis, mesmo quando o número de casos é muito grande ou infinito.

4.1 Conceitos Básicos

Para verificar formalmente um sistema, são necessários três componentes, identificados a seguir.

- Uma *linguagem de descrição do sistema*, onde é descrito o sistema computacional $\mathcal{I}$ (de software, hardware ou uma combinação de ambos) que deve ser verificado. Usualmente, esta é uma linguagem padrão de programação ou de descrição de hardware. Qualquer que seja a linguagem empregada para este fim, ela deve *necessariamente* ter uma semântica formal que indica de maneira precisa o comportamento de um sistema descrito nesta linguagem. É esta semântica que permite determinar matematicamente o comportamento que $\mathcal{I}$ apresenta.

- Uma *linguagem de especificação do sistema* que, como discutido na sessão 1.2, permite descrever o comportamento $\mathcal{P}$ esperado e as propriedades desejadas do sistema. Algumas linguagens empregadas para este fim são Z, VDM, CCS, gramáticas de grafos e diversas classes de lógicas. Assim como as linguagens de descrição do sistema, as linguagens de especificação devem ter uma semântica formal que indica precisamente o conteúdo de uma especificação.

- Um *método de inferência* que faz a ligação entre as linguagens de especificação e de descrição do sistema e permite verificar se o comportamento do sistema satisfaz a especificação apresentada. Usualmente, este método faz uma associação entre termos da linguagem de especificação e termos da linguagem de descrição de sistema que têm o mesmo significado, além de fornecer uma álgebra ou lógica para manipulação destes termos.

O quadro esboçado acima pode ser flexibilizado e estendido. A especificação de um sistema pode ser *total*, descrevendo inteiramente o comportamento desejado do sistema, ou *parcial*, descrevendo apenas algumas das propriedades que devem estar presentes em um sistema que funciona corretamente. A princípio, uma especificação total é mais interessante, por ser completa. No entanto, para sistemas reais, esta especificação pode ser muito grande, complexa ou difícil de construir ou verificar. Então se torna importante a especificação parcial de um sistema, onde se concentra o esforço em módulos ou atividades críticas do sistema e em suas principais propriedades. A verificação de especificações parciais é usualmente chamada de *verificação de propriedades*.

Como exemplo, considere o sistema de trens discutido anteriormente. Sua especificação completa é bastante grande, mas uma propriedade simples que este sistema

deve apresentar é que dois trens nunca estão no mesmo trilho. Esta propriedade simples não assegura sozinha que o sistema funciona corretamente, mas é importante porque indica que não ocorrem colisões. A verificação desta propriedade, mesmo não garantindo o total funcionamento do sistema, indica que o sistema tem condições mínimas de operar.

Não existe uma distinção intrínseca entre as linguagens de descrição e de especificação. Elas são distinguidas pelo papel que desempenham no momento da verificação formal, mas este papel pode ser trocado ao longo do processo de projeto. Um cenário onde uma mesma linguagem pode assumir diversos papeis é apresentado a seguir, onde um sistema é construído pelo refinamento sucessivo de descrições:

- Em uma primeira fase do projeto, é formulada uma descrição inicial do sistema em uma notação abstrata como uma gramática de grafos e verifica-se que esta descrição apresenta algumas propriedades críticas expressas como fórmulas em uma lógica temporal. Neste caso, a gramática de grafos desempenha o papel de uma linguagem de descrição do sistema e a lógica temporal é a linguagem empregada para a especificação parcial do sistema.

- Em uma fase seguinte do projeto, é construído um programa na linguagem C que controla o sistema de trens e verifica-se que este programa comporta-se de mesma forma que a gramática de grafos. Nesta fase, a gramática de grafos desempenha o papel de uma linguagem para especificação completa do sistema e a linguagem C descreve o sistema.

Usualmente, é importante e desejável verificar propriedades de uma descrição inicial do sistema que é usada posteriormente como sua especificação. Isto permite a detecção de erros em uma fase inicial do projeto, quando é mais fácil e barato consertá-los. Além disto, como a primeira descrição de um sistema é feita a partir de uma descrição de seu comportamento, a verificação propriedades desta especificação é a única possibilidade de verificar (ao menos de maneira limitada) que esta descrição está correta ou não contém erros graves.

É possível que a linguagem de especificação e de descrição do sistema sejam as mesmas. Um cenário onde isto ocorre é quando uma função em um programa Lisp para ordenamento de listas é substituída por outra, mais eficiente. A primeira função pode ser tratada como uma especificação para a segunda, uma vez que ambas devem comportar-se da mesma forma, e as duas funções estão codificadas na mesmo linguagem. O uso de uma mesma linguagem para especificação e descrição do sistema é conveniente porque diminui o número de formalismos envolvidos. Porém, isto não é freqüente porque a especificação e a descrição de sistemas impõem demandas muitas vezes conflitantes à linguagem.

Uma vez escolhidas as linguagens de especificação e descrição de sistemas, deve-se escolher um método de inferência para fazer a ligação entre as duas. Há duas abordagens para estabelecer esta ligação.

- Na abordagem de *modelos* para a verificação formal, a descrição do sistema é utilizada para construir um modelo $\mathcal{M}$. Este modelo é uma estrutura matemática (por exemplo, um sistema de transição finito) que representa o sistema. A verificação formal de uma propriedade $\mathcal{P}$ consiste em testar se o modelo apresenta esta propriedade, o que é normalmente denotado por $\mathcal{M} \models \mathcal{P}$. Conforme o estrutura matemática que representa o modelo e a classe de propriedades a serem verificadas, escolhe-se a técnica para realizar este teste.

- Na abordagem *dedutiva* (ou abordagem de *provas*) para a verificação formal, a descrição do sistema é utilizada para construir uma teoria $\mathcal{T}$, composta por um conjunto de fórmulas representando o sistema. A verificação formal consiste em derivar (provar) a propriedade desejada $\mathcal{P}$ a partir deste conjunto de fórmulas, indicado por $\mathcal{T} \vdash \mathcal{P}$. A linguagem escolhida para expressar as fórmulas determina a lógica a ser empregada nesta demonstração.

As duas abordagens são equivalentes do ponto de vista formal, porém elas dão origem a ferramentas distintas para o apoio à verificação formal de sistemas.

4.2 Ferramentas de Apoio

A verificação formal de sistemas deve ser apoiada por ferramentas automáticas para garantir que não há erros no próprio processo de verificação. A aplicação manual de métodos de verificação de modelos ou de demonstração de propriedades é inviável em qualquer sistema real, pois estas tarefas são repetitivas e delicadas. Para evitar erros na aplicação destes métodos, são utilizadas ferramentas específicas. Observe que a noção de erro, neste caso, são falhas na aplicação de um método de verificação. Isto não tem relação com erros de especificação ou descrição, onde as falhas são introduzidas em fases anteriores ao processo de verificação.

O nível de suporte automático que pode ser esperado na verificação formal depende da forma como o sistema é descrito e especificado. Devido a resultados teóricos fundamentais, sabe-se que, a partir de certo nível de complexidade de sistemas e de especificações, é impossível obter um suporte automático e completo ao processo de verificação. Mesmo nestes casos, no entanto, pode-se desenvolver ferramentas semi-automáticas, guiadas pelo usuário, que impedem introdução de erros durante o processo de verificação.

As principais ferramentas de apoio à verificação formal são os verificadores de modelos e os provadores de teoremas. Os *verificadores simbólicos de modelos* (symbolic model checkers) são ferramentas totalmente automáticas baseadas na abordagem de verificação de modelos. As ferramentas nesta classe usualmente tratam de sistemas que possuem um número finito de estados e verificam que este sistema satisfaz propriedades especificadas em uma lógica temporal proposicional.

Em função das restrições que os verificadores de modelos impõem às linguagens de especificação e descrição de sistemas, o processo de verificação é conduzido de forma

automática, sem a intervenção do projetista. Ao final, a ferramenta informa se o sistema apresenta ou não a propriedade considerada. Quando a propriedade não é satisfeita, a ferramenta produz, sempre que possível, uma descrição da computação que viola a propriedade.

As técnicas atuais permitem tratar de sistemas com um grande número de estados (em torno de 10^{20}). Além disto, técnicas adicionais permitem tratar classes restritas de sistemas não finitos. No entanto, a principal vantagem desta classe de ferramentas é o fato de ser totalmente automática e, para preservar este aspecto destas ferramentas, é necessário impor limitações severas na classe de propriedades e sistemas que são verificados.

Uma outra classe de ferramentas de apoio a verificação formal são os *provadores automáticos de teoremas* (automated theorem provers). Estas ferramentas seguem a abordagem dedutiva e operam sobre lógicas genéricas e expressivas, como a lógica de predicados de primeira ordem ou uma lógica alta-ordem com tipos. Devido ao poder de expressão destes formalismos, não há um método automático para provar que uma propriedade vale sobre um sistema. Neste caso, estas ferramentas utilizam heurísticas ou métodos parciais de prova, que nem sempre conseguem provar automaticamente as propriedades de um sistema.

Como os métodos empregados pelos provadores de teoremas têm limitações, sua aplicação deve ser controlada, em maior ou menor grau, pelo usuário. Ou seja, o usuário deve fornecer os principais passos que levam a verificação de uma propriedade e o sistemas realiza apenas os passos intermediários mais elementares. Porém, é a ferramenta que garante a correção da verificação formal, pois ela impede que o usuário realize passos incorretos de verificação. Isto assegura que, se uma propriedade é verificada, então não há erros neste processo e o sistema realmente apresenta esta propriedade.

Há um grande números de provadores automáticos de teoremas que diferem na lógica sobre a qual operam e no nível de suporte automático que oferecem. Algumas destas ferramentas são as comentadas a seguir:

- O provador de teoremas ACL2 [KMM00, KM97] opera sobre especificações em um sub-conjunto aplicativo de Common Lisp que constitui uma lógica de predicados de primeira ordem sem quantificadores e com indução. Este provador coordena eficientemente a aplicação de um grande número de sofisticadas regras de provas. Com isto, pode realizar provas complexas de maneira automática ou com pouco auxílio do usuário.

- O provador de teoremas PVS [Sha96] emprega uma lógica de alta-ordem com tipos bastante flexível, que simplifica a descrição de sistemas e a especificação de propriedades. Porém, em função da própria riqueza desta linguagem, seu motor de inferência é menos poderoso que o do ACL2 e a verificação de propriedades neste sistema requer mais intervenções do usuário.

- O provador de teoremas HOL [GM93] emprega uma lógica de alta-ordem com tipos e um mecanismo de prova totalmente programável e estendível. Há uma grande flexibilidade na descrição de processos de prova, mas o usuário deve selecionar, indiretamente, todo o processo de prova. O provador de teoremas Isabelle [Pau93] é uma ferramenta similar, porém parametrizável, que pode ser adaptada a qualquer lógica.

O uso adequado de um provador de teoremas requer um treinamento específico por parte do usuário. Para ser capaz de aplicar estas ferramentas na verificação de sistemas não triviais e de tamanho razoável, o usuário deve conhecer bem a arquitetura do provador e as técnicas que ele empregada na demonstração. Este treinamento pode ser longo. Porém este esforço é compensado pela flexibilidade e expressividade dos formalismos suportados por estas ferramentas, que permitem o tratamento de especificações e descrições de sistemas complexos.

Neste contexto, fica claro que os provadores de teoremas e os verificadores de modelos são ferramentas complementares. Os verificadores de modelos são utilizados preferencialmente na verificação de sistemas onde complexidade reside no controle do processamento e não na manipulação de dados. Já os provadores de teoremas são mais adequados a situações onde a manipulação de dados ou o tratamento de descrições e especificações complexas é importante.

Para continuar a discussão sobre a verificação formal, discute-se em mais detalhes os verificadores de modelos. Escolheu-se esta classe de ferramentas para este tutorial porque seu uso por um iniciante na área é mais simples. Para isto, apresenta-se lógica sobra a qual esta classe de ferramentas usualmente opera e o algoritmo de verificação que elas empregam.

4.3 Verificação de Modelos

A verificação de modelos é um procedimento para decidir se uma determinada propriedade é satisfeita (verdadeira) em uma dada estrutura matemática. Freqüentemente, a propriedade é definida através de uma fórmula de lógica temporal e corresponde a algum requisito de funcionamento do sistema sendo desenvolvido, enquanto que a estrutura matemática é algum tipo de grafo de transição, que é um modelo desse sistema.

A figura 10 representa a arquitetura típica de um sistema de verificação formal baseado em um verificador de modelos. Há diversos ferramentas de verificação de modelos disponíveis atualmente, como o SMV [McM93] e o Spin [Hol97]. Estas ferramentas são robustas e confiáveis e estão sendo aplicadas na verificação de sistemas reais tanto no ambiente acadêmico quando industrial.

A implementação do sistema a verificar (por exemplo, um programa concorrente) é traduzida em uma estrutura matemática, obedecendo à semântica da linguagem de descrição daquele sistema. Em seguida, um verificador de modelos é utilizado para

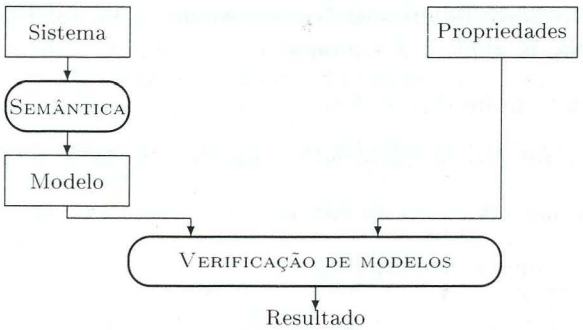

Figura 10: Ferramenta de verificação baseada em um verificador de modelos.

avaliar as diferentes propriedades de interesse. Em geral, caso a propriedade seja falsa, o verificador de modelos consegue fornecer um contra-exemplo (situação para a qual a propriedade não é válida) daquela propriedade, o que se torna extremamente útil para diagnosticar o erro que causou este problema.

Neste tutorial, focaremos mais em detalhes *a verificação simbólica de modelos*, uma técnica proposta e aprimorada durante a última década [McM93]. Neste quadro, os modelos são estruturas de Kripke codificadas através de diagramas de decisão binária (BDDs, *Binary Decision Diagrams*) e as propriedades são expressas na lógica temporal ramificada CTL (*Computation Tree Logic*).

4.3.1 Estruturas de Kripke

Estruturas de Kripke são um tipo de sistema de transição (conforme definido na seção 3.1) onde se indicam as proposições que são verdadeiras ou falsas em cada estado. Estas proposições são afirmações sobre cada estado, como "a mensagem foi enviada", "o buffer está cheio" ou "$x > 5$". Para o algoritmo de verificação de modelos, não importa o significado exato das proposições, apenas se elas são verdadeiras ou falsas em um certo estado. Portanto elas podem ser representadas simplificadamente por letras. A definição das estruturas de Kripke é apresentada a seguir.

Seja P um conjunto finito de proposições booleanas. Uma *estrutura de Kripke* sobre P é uma tupla $M = (S, T, I, L)$, onde:

- S é um conjunto finito de estados;
- $T \subseteq S \times S$ é a relação de transição tal que $\forall s \in S : \exists s' \in S : (s, s') \in T$;
- $I \subseteq S$ é o conjunto dos estados iniciais;
- $L : S \to 2^P$ é a função de etiquetagem.

Uma transição $(s, s') \in T$ indica que, durante a computação, o sistema pode passar do estado s para o estado s'. A restrição sobre esta relação exige que cada estado tenha ao menos um estado seguinte, ou seja, o sistema não pára nunca. Por fim, a função de etiquetagem L associa cada estado com o conjunto das proposições booleanas que são verdadeiras neste estado.

Exemplo 4 *Considere o conjunto das proposições atômicas $P = \{g, s, w, b\}$. A Figura 11 apresenta uma estrutura de Kripke que emprega estas proposições. Os nodos e arcos deste diagrama representam os estados e as transições de estado. Cada nodo é anotado com o nome do estado que ele representa e com o conjunto das proposições atômicas que são verdadeiras neste estado. Esta estrutura de Kripke é definida como $A = (S_A, T_A, I_A, L_A)$ onde:*

$$
\begin{aligned}
S_A &= \{s_0, s_1, s_2, s_3, s_4, s_5\} \\
T_A &= \{\ (s_0, s_0), (s_0, s_1), (s_1, s_2), (s_2, s_1), (s_2, s_3), \\
&\qquad (s_3, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_4), (s_5, s_0)\}; \\
I_A &= \{s_0, s_3\}; \\
L_A &= \{s_0 \mapsto \{g\}, s_1 \mapsto \{s\}, s_2 \mapsto \{w\}, s_3 \mapsto \{g, b\}, s_4 \mapsto \{s, b\}, s_5 \mapsto \{w, b\}\}.
\end{aligned}
$$

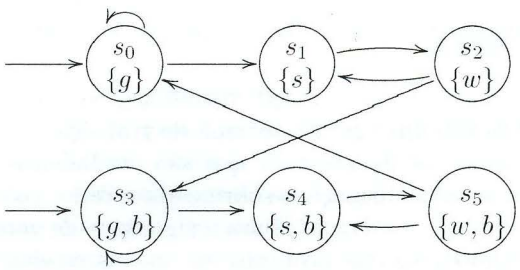

Figura 11: Diagrama de transição de uma estrutura de Kripke

Estruturas de Kripke formam um modelo de computação bastante adequado para modelar sistemas reativos, protocolos, *hardware*, etc. Porém, sua representação consiste em enumerar cada estado e transição da estrutura, e o preço de armazenamento é então função do tamanho de S e T.

Uma alternativa consiste em representar e manipular *conjuntos* de estados e transições através de suas funções características, que no caso de estruturas de Kripke são termos da lógica proposicional. A função característica de um estado $s \in S$, notada $[s]$, é definida como:

$$[s](\mathbf{v}) = \left(\left(\bigwedge_{v_i \in L(s)} v_i \right) \wedge \left(\bigwedge_{v_i \notin L(s)} \neg v_i \right) \right)$$

Essa definição é estendida a conjuntos de estados, utilizando as seguintes regras:

$$\begin{aligned} [\emptyset](\mathbf{v}) &= false \\ [\{x\} \cup X](\mathbf{v}) &= [x](\mathbf{v}) \vee [X](\mathbf{v}) \end{aligned}$$

Exemplo 5 (Caracterização de estados) *Na estrutura de Kripke A do exemplo anterior, temos as seguintes funções características:*

$$\begin{aligned} [s_0] &= g \wedge \neg s \wedge \neg w \wedge \neg b \\ [I] &= g \wedge \neg s \wedge \neg w \end{aligned}$$

Assim, o valor da variável b não importa para a caracterização do conjunto de estados iniciais, desaparecendo da fórmula.

Para representar transições, precisamos introduzir um novo conjunto de variáveis booleanos $P' = \{v'_1, \ldots v'_n\}$. P' corresponde aos valores das variáveis no estado de chegada da transição. A função característica de uma transição $t = (s_1, s_2) \in T$, notada $[t]$, é definida como:

$$[t](\mathbf{v}, \mathbf{v}') = [s_1](\mathbf{v}) \wedge [s_2](\mathbf{v}')$$

Essa definição pode ser estendida à representação de conjuntos de transições como fizemos no caso de estados.

Exemplo 6 (Caracterização de transições) *Na estrutura A do exemplo 4, temos as seguintes funções características ($\underline{\vee}$ é o ou exclusivo):*

$$\begin{aligned} [(s_0, s_1)] &= (g \wedge \neg s \wedge \neg w \wedge \neg b) \wedge (\neg g' \wedge s' \wedge \neg w' \wedge \neg b') \\ [T] &= \quad (b \underline{\vee} b') \wedge \neg g \wedge \neg s \wedge w \wedge g' \wedge \neg s' \wedge w' \\ &\quad \vee \ (b \leftrightarrow b') \wedge \quad (g \wedge \neg s \wedge \neg w \wedge \neg w' \wedge (g' \underline{\vee} s')) \\ &\qquad\qquad \vee \ (\neg g \wedge s \wedge \neg w \wedge \neg g' \wedge \neg s' \wedge w') \\ &\qquad\qquad \vee \ (\neg s \wedge \neg g' \wedge s' \wedge \neg w' \wedge (g \underline{\vee} w)) \end{aligned}$$

4.3.2 Lógica CTL

Numa estrutura de Kripke, uma seqüência de estados corresponde a uma possível computação do sistema. Como, em cada estado, várias transições são geralmente possíveis, cada possibilidade de computação forma um ramo de uma árvore de computação infinita. Essa árvore de computação representa todas as computações possíveis do modelo. CTL (*Computation Tree Logic* [CE81]) é uma lógica para raciocinar sobre estruturas de Kripke e que é interpretada sobre as árvores de computação.

Fórmulas da lógica CTL são construídas a partir das proposições atômicas da estrutura de Kripke combinadas através dos operadores booleanos tradicionais e dos operadores temporais. Por exemplo, a fórmula $\mathbf{EG}(\neg s \wedge \neg w)$ é construída a partir das proposições s e w, combinadas com os operadores booleanos $\neg$ e $\wedge$ e o operador temporal **EG**. Em CTL, cada operador temporal tem dois elementos:

- O *quantificador de caminhos* indica sobre quais caminhos da computação a fórmula-argumento deve ser avaliada:
  - o quantificador **E** indica que o argumento deve ser verdadeiro em *algum* caminho da computação;
  - o quantificador **A** indica que o argumento deve ser verdadeiro em *todos* os caminhos.
- O *quantificador de estados* (que sempre segue um quantificador de caminhos) indica sobre quais estados do caminho considerado a fórmula-argumento deve ser avaliada:
  - o quantificador **X** indica que o argumento deve ser verdadeiro no *próximo* estado do caminho considerado;
  - o quantificador **G** indica que o argumento deve ser verdadeiro em *todos* os estados deste caminho;
  - o quantificador **F** indica *algum* estado do caminho considerado;
  - o quantificador **U** indica explicitamente a condição até quando é avaliado o argumento.

Por exemplo, a fórmula $\mathbf{EG}(\neg s \wedge \neg w)$ é verdadeira se existe uma computação (**E**), onde todos os estados (**G**) satisfazem a condição $\neg s \wedge \neg w$. Dois operadores temporais de destaque são **AG** ("para todos os caminhos, para todos os estados" ou seja "sempre") e **AF** ("para todos os caminhos, em algum estado" ou seja "futuramente").

Embora CTL apresente algumas restrições teóricas em relação a seu poder de expressão, na prática, quase todas as propriedades interessantes podem ser escritas como uma fórmula de CTL. Seguem alguns exemplos de propriedades freqüentemente encontradas:

- $\mathbf{AG}\neg(ack_1 \wedge ack_2)$ exprime a exclusão mútua de ack_1 e ack_2. Ou seja, em todos os estados de todos os caminhos de computação, ack_1 e ack_2 nunca são verdadeiros simultaneamente.

- $\mathbf{AG}(req \rightarrow \mathbf{AF}ack)$ diz que cada vez que req é verdadeiro, então ack virá obrigatoriamente a ser verdadeiro. Ou seja, em todo o caminho da computação, os estados onde req é verdadeiro são sempre seguidos por algum estado futuro onde ack torna-se verdadeira. Informalmente, isto significa que toda a requisição é atendida.

4.3.3 Algoritmo de Verificação

Para realizar a verificação de modelos de sistemas com um grande número de estados e transições é preciso ter uma representação muito eficiente para a lógica proposicional, que é a forma como são codificadas as estruturas de Kripke. Atualmente a representação que se mostrou a mais eficiente são os BDDs (*Binary Decision Diagrams*) [Bry86], que possui uma representação relativamente compacta e permite realizar operações como a conjunção e a disjunção em tempo linear.

Uma vez construídos os BDDs do modelo, é preciso verificar cada uma das propriedades da especificação. A *verificação simbólica de modelos* é o algoritmo que realiza este papel. Dados uma propriedade, este algoritmo retorna a função característica dos estados da estrutura que satisfazem aquela propriedade. Se os estados iniciais fazem parte deste conjunto, então a propriedade é verdadeira, caso contrário, ela é falsa e é procurado um contra-exemplo.

Basicamente, a verificação simbólica de modelos consiste em cálculos de pontos fixos de conjuntos de estados. Por exemplo, para verificar quais são os estados que satisfazem $\mathbf{EG}f$, o objetivo é calcular a função característica do conjunto de estados onde f é verdadeira e tal que existe um estado sucessor onde $\mathbf{EG}f$ é verdadeiro. O algoritmo realizando esta tarefa tem como parâmetro a função característica dos estados onde f é verdadeira e retorna a função característica dos estados onde $\mathbf{EG}f$ é verdadeira (figura 12). A variável *res* acumula os valores sucessivos necessários para alcançar o ponto fixo.

A função *Predecessor* é usada para calcular a função característica de todos os predecessores de um conjunto de estados, dada sua função característica F. Esse resultado é obtido avaliando a fórmula $\forall v : \exists v' : T(v, v') \wedge F(v')$, onde T é a função característica de todas as transições da estrutura. Todas essas operações podem ser realizadas com BDDs.

Este algoritmo de verificação simbólica de modelos forma a base de ferramentas que foram aplicadas com sucesso a diversos tipos de sistemas tanto hardware [CGH+93, MS92] como software [ABB+96]. Esta técnica ainda hoje é sendo objeto de inúmeras pesquisas para aperfeiçoar seu desempenho e estender sua atuação a novos domínios de atuação. Com BDDs, é comum representar e verificar sistemas com

```
1   algoritmo ModelCheckEG(f: BDD) : BDD
2   var
3       res, tmp: BDD
4   início
5   res ← f
6   faça
7       tmp ← res
8       res ← BddAnd(f, Predecessor(tmp))
9   enquanto tmp ≠ res
10  retorna res
11  fim ModelCheckEG
```

Figura 12: Algoritmo para verificar propriedades **EG**

mais de 10^{20} [BCM+90] estados de forma totalmente automática. Para poder tratar sistemas de maior porte, é preciso integrar a verificação simbólica de modelos com outras técnicas, como abstração e a verificação composicional. O leitor interessado em aprofundar-se no estudo da verificação de modelos pode consultar [CGP00], que apresenta o estado da arte desta técnica.

5. Conclusão

Este tutorial apresentou uma introdução aos métodos formais aplicados a verificação de sistemas concorrentes. Discutiram-se formalismos para descrever matematicamente a semântica destes sistemas, bem como métodos e técnicas para especificação e verificação formal. Foram estudados com mais detalhes as gramáticas de grafos, os sistemas de transição e os verificadores de modelos.

O objetivo de empregar métodos formais no desenvolvimento de sistemas de software ou hardware é melhorar a qualidade do produto. Ou seja, construir sistemas mais confiáveis, em um prazo menor e com um custo menor. Atualmente, o maior empecilho a aplicação destes métodos é a falta de treinamento dos projetistas de sistemas computacionais. Estes projetistas desconhecem estas técnicas e, além disto, acreditam incorretamente que estas técnicas são "difíceis", "caras", "não são práticas" e "consomem muito tempo". Para superar estas barreiras, é preciso divulgar os conceitos e técnicas básicas entre os profissionais da área, informando como e quando aplicá-las e mostrando com evidências práticas que elas trazem benefícios concretos a projetos reais. Este tutorial é uma contribuição para esta formação básica sobre métodos formais.

Neste tutorial, a discussão concentrou-se em sistemas concorrentes porque é nesta área que o benefício do emprego de métodos formais pode ser melhor apreciado. O

projeto de sistemas concorrentes é complexo, caro e lento. As técnicas usuais de projeto não se mostram capazes de administrar esta complexidade. Neste caso, o emprego de técnicas inovadoras como os métodos formais torna-se atraente. De fato, a precisão de descrições, a ausência de ambigüidades e a possibilidade de tratar de um número muito grande ou infinito de casos são essenciais no projeto de um sistema concorrente. Com a emergência de sistemas altamente distribuídos e concorrentes como a Internet ou os modernos chips, o projeto de sistemas concorrentes não é mais uma atividade de poucos, mas algo que todo o profissional da área deve ser capaz de realizar. Neste contexto, os métodos formais tornam-se uma opção prática e relevante.

Os métodos formais, porém, não são um produto acabado, onde tudo está feito. Pelo contrário, é uma área de pesquisa constante, onde a evolução é rápida. Atualmente, grandes esforços concentram-se no desenvolvimento de novos métodos de especificação e verificação que são aplicados mais facilmente ou são adequados a um certo domínio de aplicação. São aperfeiçoadas as técnicas de verificação, permitindo o tratamento de sistemas maiores e mais complexos. Surgem novas abordagens para a descrição de sistemas que sugerem novas abordagens para processo de desenvolvimento de sistemas. Pesquisam-se novos modelos matemáticos que consideram novos aspectos dos sistemas, como segurança, tolerância a falhas, tempo real, interfaces flexíveis, etc. O que foi apresentado neste tutorial é apenas uma pequena parte de tudo isto, mas espera-se que ela tenha apresentado um bom retrato deste campo e que o leitor ou leitora motive-se para continuar seu estudo na área. As referências apresentadas ao longo do texto são um ponto de partida para este estudo.

Referências

[ABB+96] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and R. Reese, *Model checking large software specifications*, 4^{th} Symposium on the Foundations of Software Engineering, ACM/SIGSOFT, October 1996, pp. 156–166.

[Abr85] J. R. Abrial, *Programming as a mathematical exercise*, Mathematical logic and programming languages (C. A. R. Hoare, ed.), Prentice-Hall International, 1985.

[BB87] Tommaso Bolognesi and Ed Brinksma, *Introduction to the Iso specification language LOTOS*, Computer networks and ISDN systems **14** (1987), no. 1.

[BBFM82] H. C. Berg, W. E. Boebert, W. R. Frante, and T. G. Moher, *Formal methods of program verification and specification*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[BCM+90] Jerry R. Burch, Edmund M. Clarke Jr, Kenneth L. McMillan, David L. Dill, and J. Hwang, 10^{20} *states and beyond*, LICS'90: 5^{th} annual IEEE

symposium on logic in computer science (Philadelphia, PA), IEEE, 1990, pp. 428–439.

[BH95] J. P. Bowen and M. G. Hinchey, *Seven more myths of formal methods*, IEEE Software (1995).

[BJ78] D. Bjørner and C. B. Jones (eds.), *The vienna development method: the meta-language*, Lecture Notes in Computer Science, vol. 61, Springer-Verlag, Berlim, 1978.

[Bry86] Randy E. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **C-35** (1986), no. 12, 205–213.

[BW90] M. Barr and C. Wells, *Category theory for computing science*, Series in Computer Science, Prentice Hall International, London, 1990.

[CE81] Edmund M. Clarke Jr and E. Allen Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, Logic of Programs: Workshop (Yorktown Heights, NY), Lecture Notes in Computer Science, no. 131, Springer Verlag, 1981.

[CGH+93] Edmund M. Clarke Jr, Orna Grumberg, H. Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and L.A. Ness, *Verification of the futurebus+ cache coherence protocol*, 11th International Symposium on Computer Hardware Description Languages: CHDL'93 (L. Claesen, ed.), North-Holland, 1993.

[CGP00] Edmund M. Clarke Jr, Orna Grumberg, and Doron A. Peled, *Model checking*, M.I.T. Press, 2000.

[CMR96] A. Corradini, U. Montanari, and F. Rossi, *Graph processes*, Fundamenta Informaticae **26** (1996), no. 3/4, 241–265.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe, *Algebraic approaches to graph transformation I: Basic concepts and double pushout approach*, The Handbook of Graph Grammars, Volume 1: Foundations, World Scientific, 1997, pp. 163–246.

[CoF99] CoFI Group, *Casl — the common algebraic specification language*, FM'99 — World Congress on Formal Methods, Springer Verlag, 1999, eletronic media.

[Cor95] A. Corradini, *Concurrent computing: from Petri nets to graph grammars*, Eletronic Notes in Theoretical Computer Science **2** (1995), 245–260, Proc. of the SEGRAGRA'95 Workshop on Graph Rewriting and Computation.

[CW96] E. Clarke and J. Wing, *Formal methods: State of the art and future directions*, Tech. Report cmu-cs-96-178, Carnegie Mellon University, 1996.

[DF98] R. Diaconecu and K. Futatsugi, *CafeOBJ report: The language, proof techniques, and methodologies for object oriented algebraic specification*, AMAST Series in Computing, vol. 6, World Scientific, 1998.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini, *Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach*, The Handbook of Graph Grammars, Volume 1: Foundations, World Scientific, 1997, pp. 247–312.

[Ehr79] H. Ehrig, *Introduction to the algebraic theory of graph grammars*, 1st Graph Grammar Workshop, Lecture Notes in Computer Science 73 (V. Claus, H. Ehrig, and G. Rozenberg, eds.), Springer Verlag, 1979, pp. 1–69.

[EKR91] H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *4th International Workshop on Graph Grammars and Their Application to Computer Science*, Springer Verlag, 1991, Lecture Notes in Computer Science 532.

[Fis96] C. Fisher, *Combining CSP and Z*, Tech. report, University of Oldenburg, 1996.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing, *The Larch family of specification languages*, IEEE Software **2** (1985), no. 5.

[GM93] M. J. C. Gordon and T. F. Melham (eds.), *Introduction to HOL; a theorem proving environment for higher order logic*, Cambridge University, Cambridge, 1993.

[GT86] Joseph A. Goguen and Joseph J. Tardo, *An introduction to Obj*, Software specification techniques (N. Gehani and A. D. Mcgettrick, eds.), Addison-Wesley Publishing Company, 1986, pp. 391–419.

[GTW77] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, *An initial algebra approach to the specification, correctness and implementation of abstract data types*, Tech. Report RC 6487 (26817), IBM Thomas J. Watson Research Center, Yorktown Heights, New York, out. 1977.

[Hal90] J. A. Hall, *Seven myths of formal methods*, IEEE Software (1990).

[Hoa85a] C. A. R. Hoare, *Communicating sequential processes*, Prentice-Hall International, London, 1985.

[Hoa85b] C. A. R. Hoare, *Communicating sequential processes*, International Series in Computer Science, Prentice Hall, London, 1985.

[Hol97] Gerard J. Holzmann, *The spin model checker*, IEEE Trans. on Software Engineering (1997).

[KM97] Matt Kaufmann and J S. Moore, *An industrial strength theorem prover for a logic based on Common Lisp*, IEEE Transactions on Software Engineering **23** (1997), no. 4, 203–13.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J S. Moore, *Computer-aided reasoning: An approach*, Kluwer, 2000.

[Kor94a] M. Korff, *Graph-interpreted graph transformations for concurrent object-oriented systems*, Extended abstract for the 5th International Workshop on Graph Grammars and their Application to Computer Science, 1994.

[Kor94b] M. Korff, *True concurrency semantics for single pushout graph transformations with applications to actor systems*, Working papers of the International Workshop on Information Systems – Corretness and Reusability IS-CORE'94, 1994, Tech. Report IR-357, Free University, Amsterdam., pp. 244–258.

[Kor96] M. Korff, *Generalized graph structure grammars with applications to concurrent object-oriented systems*, Ph.D. thesis, Technical University of Berlin, 1996.

[KR96] M. Korff and L. Ribeiro, *Formal relationship between graph grammars and Petri nets*, Lecture Notes in Computer Science **1073** (1996), 288–303.

[KR98] M. Korff and L. Ribeiro, *True concurrency = interleaving + weak conflict*, Eletronic Notes in Theoretical Computer Science **14** (1998).

[LB86] Barbara Liskov and Valdis Berzins, *An appraisal of software specifications*, Software specification techniques (N. Gehani and A. D. Mcgettrick, eds.), Addison-Wesley Publishing Company, 1986, pp. 3–22.

[LKW93] M. Löwe, M. Korff, and A. Wagner, *An algebraic framework for the transformation of attributed graphs*, Term Graph Rewriting: Theory and Practice (M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, eds.), John Wiley & Sons Ltd, 1993, pp. 185–199.

[Löw93] M. Löwe, *Algebraic approach to single-pushout graph transformation*, Theoretical Computer Science **109** (1993), 181–224.

[Man74] Zohar Manna, *Mathematical theory of computation*, McGraw-Hill Computer Science Series, McGraw-Hill Inc., EUA, 1974.

[Mar88] Raul C. B. Martins, *Sistemas formais e desenvolvimento de software*, Tech. Report CCR 064, Centro Científico Rio - IBM Brasil, Rio de Janeiro, Brasil, nov. 1988.

[McM93] Kenneth L. McMillan, *Symbolic model checking: an approach to the state-space explosion problem*, Kluwer Academic Publisher, 1993.

[Mil80] R. Milner, *A calculus of communicating systems*, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, Berlim, 1980.

[MS92] Kenneth L. McMillan and J. Schwalbe, *Shared memory multi-processing*, ch. Formal Verification of the Gigamax Cache Coherency Protocol, MIT Press, 1992.

[Nag87] M. Nagl, *A software development environment based on graph technology*, 3rd Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science 291 (H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, eds.), Springer Verlag, 1987, pp. 458–478.

[Nag92] M. Nagl, *The use of graph grammars in applications*, Graph Grammars and Their Application to Computer Science, Springer Verlag, 1992, Lecture Notes in Computer Science 532, pp. 41–60.

[Pag81] Frank G. Pagan, *Formal specifiction of programming languages: a panoramic primer*, Prentice-Hall, Inc., New Jersey, 1981.

[Pau93] Lawrence C. Paulson, *Isabelle: A generic theorem prover*, Springer-Verlag, 1993, LNCS, 828.

[Rib96] L. Ribeiro, *Parallel composition and unfolding semantics of graph grammars*, Ph.D. thesis, Technical University of Berlin, Germany, 1996.

[Roz87] G. Rozenberg, *An introduction to the NLC way of rewriting graphs*, 3rd Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science 291, Springer Verlag, 1987, pp. 55–66.

[Sha96] N. Shankar, *PVS: Combining specification, proof checking and model checking*, FMCAD'96, 1996, LNCS 1166, pp. 257–264.

[SNW93] V. Sassone, M. Nielsen, and G. Winskel, *A classification of models for concurrency*, 4th International Conference on Concurrency Theory, Springer, 1993, Lecture Notes in Computer Science 715, pp. 82–96.

[SRI98] SRI INternational, *Maude manual*, 1998, available at http://maude.csl.sri.com.

[Sta89] A. Stark, *Concurrent transition systems*, Theoretical Computer Science **64** (1989), 221–269.

[Sto77] J. E. Stoy, *Denotational semantics: The scott-strachey approach to programming language theory*, MIT press, Cambridge, Massachusets, 1977.

[Win85] G. Winskel, *Synchronization trees*, Theoretical Computer Science (1985), no. 34, 33–82.

[Win89] G. Winskel, *An introduction to event structures*, Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Springer Verlag, 1989, pp. 364–397.