Uma Metodologia para a Verificação de Sistemas Parciais Modelados na Gramática de Grafos Baseada em Objetos[*]

Fernando L. Dotti, Fábio Pasini, Osmar M. dos Santos[**]
Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, RS - Brasil
e-mail: [fldotti, fpasini, osantos]@inf.pucrs.br

Resumo

O desenvolvimento de sistemas distribuídos é considerado uma tarefa complexa. Em especial, garantir que estes sistemas apresentam certas propriedades, fundamentais para o seu correto funcionamento, torna-se muito difícil em ambientes abertos, como, por exemplo, a Internet. Neste artigo utilizamos uma linguagem visual de especificação formal, chamada Gramática de Grafos Baseadas em Objetos (GGBO), para a especificação e verificação de sistemas distribuídos. É proposta uma metodologia para a verificação de sistemas parciais, baseada no raciocínio assume-guarantee, definidos na GGBO. Na metodologia proposta, uma transformação é realizada sobre o objeto a ser verificado. A transformação tem como objetivo fechar (completar) o comportamento do objeto, de forma que a técnica de verificação de modelos possa ser aplicada ao mesmo. Um objeto que implementa a ordenação de mensagens é utilizado, no decorrer do artigo, para ilustrar a metodologia definida.

Abstract

The development of distributed systems is considered a complex task. In particular, the process of assuring certain properties of the system, necessary for its correct execution, becomes really difficult in open environments, like the Internet. In this paper, we use a visual formal specification language, called Object-Based Graph Grammars (OBGG), for the specification and verification of distributed systems. A methodology for the verification of partial systems, using the assume-guarantee reasoning, defined in OBGG is proposed. In this methodology, a transformation is applied over the object to be verified. The transformation is used to close the behavior of the object in such way that model checking techniques can be applied to the object. As a running example, an object implementing message ordering is used to illustrate the defined methodology.

1. Introdução

O desenvolvimento de sistemas distribuídos é considerado uma tarefa complexa. Em especial, garantir que estes sistemas apresentam certas propriedades, fundamentais para o seu correto funcionamento, torna-se muito difícil em ambientes abertos, como, por exemplo, a Internet. Ambientes abertos permitem formas variadas de cooperação e integração entre organizações e sistemas, sendo caracterizados por [1]: alta distribuição geográfica, ultrapassar fronteiras administrativas; dinamismo (oferta e retirada de serviços e nós computacionais); inexistência de controle global; falhas parciais; falta de segurança;

[*] Este trabalho foi parcialmente desenvolvido em colaboração com a HP Brasil P&D. Este trabalho é parcialmente financiado pelo projeto de pesquisa IQ-Mobile (CNPq/CNR).
[**] Autores aparecem em ordem alfabética.

heterogeneidade tanto de nós computacionais (capacidade de processamento), como enlaces de comunicação (atraso, vazão e perda de pacotes); falta de qualidade na comunicação. Com esta variedade de aspectos envolvidos muitas vezes é difícil determinar se um comportamento indesejado é originado pelo sistema em desenvolvimento ou pelo ambiente em que ele executa.

Desta forma, faz-se necessário que desenvolvedores de sistemas distribuídos tenham maior confiança em suas soluções durante a fase de desenvolvimento do sistema. Nesse sentido, nosso grupo de pesquisa tem desenvolvido métodos e ferramentas para auxiliar desenvolvedores na construção de sistemas distribuídos, sendo que o uso de métodos formais provê uma forma precisa de modelar e analisar o sistema em desenvolvimento. Mais especificamente, em [2] a linguagem de especificação formal Gramática de Grafos Baseada em Objetos (GGBO) foi proposta. Essa linguagem apresenta abstrações básicas que a tornam apropriada para a especificação de sistemas distribuídos assíncronos.

Sistemas modelados na GGBO podem ser analisados através de simulação [3,4] e, a partir de resultados mais recentes, através de verificação de modelos [5,6]. Além da possibilidade de analisar sistemas distribuídos modelados na GGBO, também é possível gerar código para execução em um ambiente real, seguindo uma tradução de GGBO para código Java [4].

Neste artigo iremos nos deter na análise de modelos GGBO através de verificação de modelos. Mais especificamente, uma metodologia para a verificação de sistemas parciais modelados na GGBO é proposta. Como pode ser visto na literatura, vários são os trabalhos que objetivam garantir propriedades sobre partes de um sistema, ao invés do sistema inteiro, usando a técnica *assume-guarantee* em conjunto da verificação de modelos [7,8]. Em comum, estes trabalhos apresentam a necessidade de fechar (completar) o comportamento da parte do sistema que será verificada, de forma que ferramentas de verificação de modelos possam ser aplicadas na análise. Por fechar o comportamento de uma parte do sistema entende-se uma transformação em que comportamentos do ambiente, necessários para o funcionamento desta parte do sistema, são adicionados à parte do sistema criando um módulo, que será posteriormente verificado através de ferramentas de verificação de modelos.

Seguindo o mesmo raciocínio, neste artigo a metodologia de verificação de sistemas parciais modelados na GGBO tem como base fechar o comportamento do objeto a ser verificado. Após o fechamento do objeto a ser analisado, o raciocínio *assume-guarantee* [9] é usado para verificação. Dessa forma, é possível garantir que o objeto irá apresentar certas propriedades necessárias para o seu correto funcionamento (*guarantee*), assumindo que o ambiente se comporta de determinada forma (*assume*). Além do seu uso na verificação de ambientes abertos distribuídos, entre as principais motivações para o desenvolvimento dessa metodologia estão características inerentes da linguagem GGBO, como ser baseada em objetos e reativa, facilitando a aplicação da técnica *assume-guarantee*.

Este artigo é organizado da seguinte forma: a Seção 2 apresenta a linguagem de especificação formal GGBO em conjunto de um exemplo de modelagem, sendo brevemente revisado o método de verificação de modelos existente para a GGBO; na Seção 3 é visto o estado da arte na área de verificação de sistemas parciais utilizando verificação de modelos, e é proposta a metodologia para a verificação de sistemas parciais modelados na GGBO; a Seção 4 apresenta trabalhos relacionados encontrados na literatura; na Seção 5 são expostas as considerações finais sobre a metodologia definida e os principais trabalhos futuros.

2. Gramática de Grafos Baseada em Objetos

As gramáticas de grafos [10] fornecem uma forma bastante natural de expressar situações complexas, onde os estados do sistema são descritos por grafos e os aspectos dinâmicos podem ser capturados pelas regras da gramática. Uma gramática de grafos é composta por:

- Um grafo de tipos, que representa os tipos dos vértices e arestas permitidas no sistema;
- Um grafo inicial, que representa o estado inicial do sistema;
- Um conjunto de regras que descrevem as possíveis mudanças de estado que podem ocorrer em um sistema.

Em [2] é proposta uma restrição de gramática de grafos, chamada GGBO (Gramática de Grafos Baseada em Objetos), para descrever sistemas baseados em objetos. Na GGBO um sistema consiste de entidades autônomas, chamadas de objetos. Os objetos possuem um estado interno e se comunicam através da troca de mensagens. Graficamente um objeto tem a notação de um retângulo, onde consta seu nome, um número (que identifica o tipo de objeto), e o conjunto de atributos. Atributos de tipos de dados pré-definidos são listados dentro do retângulo, sendo que os atributos que referenciam outros objetos têm a notação de arestas que se ligam ao número do tipo de objeto (ver Figura 1).

O comportamento de um objeto corresponde às reações executadas por ele ao receber mensagens. Estas reações podem mudar o estado interno do objeto e/ou causar o envio de mensagens para outros objetos ou para si mesmo. As mensagens devem ter como destino um objeto e podem ter como argumentos outros objetos ou valores de tipos pré-definidos. Graficamente uma mensagem tem a forma de um polígono como o da mensagem *Msg* na Figura 1. O destino desta mensagem é dado por uma seta e os vários argumentos da mensagem são dados por linhas que ligam estes argumentos ao polígono da mensagem.

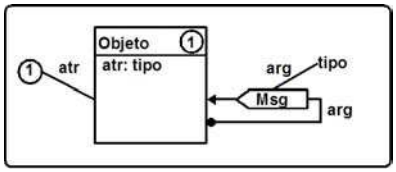

Figura 1. Grafo modelo para sistemas baseados em objetos.

Em um sistema baseado em objetos podem existir vários tipos de objetos (também chamados de entidades). Para descrever um sistema baseado em objetos usando GGBO é necessário definir os grafos de tipos dos objetos, que descrevem os tipos de objetos que compõem o modelo. O grafo de tipos de uma entidade define os atributos que um objeto tem e os tipos de mensagens que um objeto daquele tipo pode receber, bem como seus parâmetros (argumentos). Como exemplo, vide Figuras 2 (a) e 3 (a).

O grafo inicial de um modelo descreve todos os objetos, mensagens e valores de atributos que devem existir na situação inicial desejada do modelo. Estes objetos com seus atributos e mensagens são instâncias dos tipos de objetos (entidades) do grafo de tipos. Os objetos podem ser instanciados estaticamente (no grafo inicial) ou dinamicamente (através da execução de regras). Como exemplo de grafo inicial vide Figura 4.

Por fim, as regras modelam o comportamento de uma entidade. Em uma gramática de grafos uma regra $r: L \rightarrow R$ especifica uma mudança de estado do sistema que ocorre da seguinte forma:

- Todos os itens que estão no lado esquerdo da regra L devem estar presentes no estado atual do sistema para possibilitar a aplicação da regra;
- Todos os itens que são mapeados de L para R (através do mapeamento r) são preservados;

- Todos os itens que não são mapeados de *L* para *R* são apagados do estado atual;
- Todos os itens que estão em *R* e não estão em *L* são adicionados para a obtenção do novo estado.

As regras de uma entidade são as que apresentam, no lado esquerdo da regra, uma mensagem sendo recebida por um objeto do tipo em questão. Esta regra especifica a reação de um objeto daquele tipo à recepção daquela mensagem. No lado direito da regra, esta mensagem terá sido consumida, atributos do objeto podem ter mudado de valor, e novas mensagens podem ter sido geradas. Cada regra descreve o tratamento de apenas uma mensagem. Todas as ações descritas em uma regra ocorrem de forma atômica. Como exemplo, vide Figuras 2 (b) e 3 (b).

Uma GGBO fornece uma linguagem baseada em objetos para a especificação de sistemas. Essas especificações apresentam algumas características que facilitam o desenvolvimento do sistema que descrevem. Um sistema baseado em objetos é modular, uma vez que é composto por entidades autônomas (objetos) conectados via interfaces bem definidas (mensagens). Os atributos e as operações que manipulam os objetos são descritos juntos no objeto e não podem ser acessados por outros objetos (encapsulamento).

Além disso, a GGBO permite modelar concorrência e não-determinismo. A concorrência entre objetos e a concorrência interna (um objeto pode tratar diversas mensagens ao mesmo tempo) são modeladas pela possibilidade da aplicação de diversas regras ao mesmo tempo, quando elas não são conflitantes. Uma regra r_1 é conflitante com uma regra r_2 se consomem (apagam) alguns itens em comum. O não-determinismo é modelado na escolha da regra para aplicação, isto é, se mais de uma regra puder ser aplicada em uma situação, uma dessas é escolhida não-deterministicamente para executar. A recepção de mensagens na GGBO também ocorre de forma não-determinística.

2.1. Exemplo

Em ambientes abertos, podem existir casos em que mensagens chegam a um nodo destino de forma não-ordenada. Na GGBO, essa característica do ambiente é naturalmente modelada, pois a recepção de mensagens ocorre de forma não-determinística na linguagem. Com o objetivo de ordenar mensagens em um nodo destino, nesta seção é apresentada a modelagem de um objeto (Figura 3) que implementa a ordenação de mensagens recebidas e as repassa para um objeto representando o usuário (Figura 2).

O modelo definido nessa seção é composto por duas entidades (tipos de objetos): *User* e *Node*. Um objeto do tipo *User* é composto por (Figura 2 (a)): uma referência para um objeto do tipo *Node* que implementa a ordenação de mensagens (atributo *node*), um identificador (atributo *id*), um contador indicando o número de mensagens já geradas (atributo *msg_g*), e o número máximo de mensagens que um objeto pode gerar até que novas mensagens sejam recebidas do destino (atributo *msg_n*). Além disso, um objeto do tipo *User* pode receber dois tipos de mensagens: *GenMsg* e *Msg*. O comportamento de um objeto do tipo *User* é cíclico, como pode ser visto na Figura 2 (b). Ou seja, o objeto fica gerando continuamente um número máximo de mensagens (regra *Generate*). Quando mensagens ordenadas são repassadas pelo objeto de tipo *Node*, elas são consumidas pelo *User* (regras *Consume1* e *Consume2*). Ao consumir as mensagens, caso o número de mensagens já geradas seja igual a zero (nenhuma mensagem foi gerada ainda), o objeto simplesmente consome a mensagem (regra *Consume1*). Caso contrário, o número de mensagens geradas for maior que zero, o objeto consome a mensagem e decrementa o número de mensagens já geradas (regra *Consume2*), habilitando a geração de novas mensagens (aplicação da regra *Generate*).

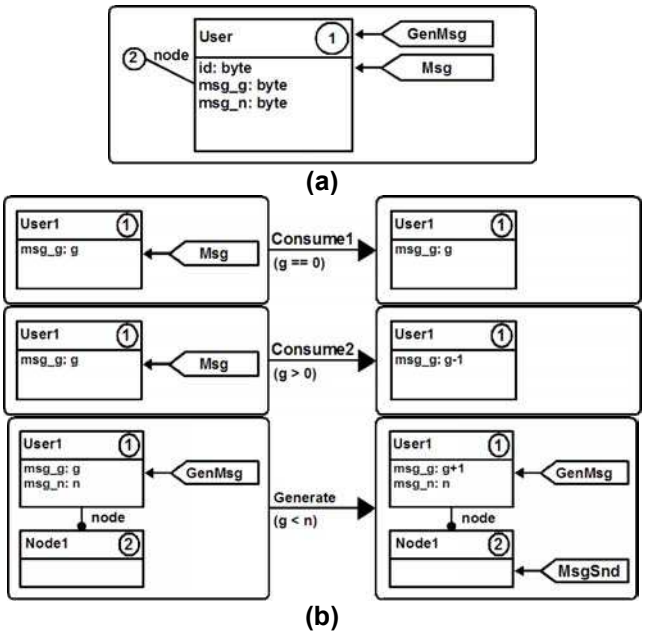

Figura 2. Grafo de tipos (a) e regras (b) para a entidade *User*.

Um objeto do tipo *Node* é composto por (Figura 3 (a)): uma referência para um objeto do tipo *User* que gera novas mensagens a serem enviadas e, ao mesmo tempo, é o receptor das mensagens ordenadas (atributo *user*), uma referência para um objeto do tipo *Node* para onde as mensagens são enviadas (atributo *node*), um identificador (atributo *id*), um contador indicando o número de seqüência da próxima mensagem a ser enviada (atributo *seq_s*), um contador indicando o número de seqüência da mensagem esperada (atributo *seq_e*), um atributo utilizado para tornar visível, na verificação, o número de sequência das mensagens recebidas e das mensagens processadas (atributo *seq_rp*), e um valor utilizado para gerar números de seqüência (atributo *msg_n*).

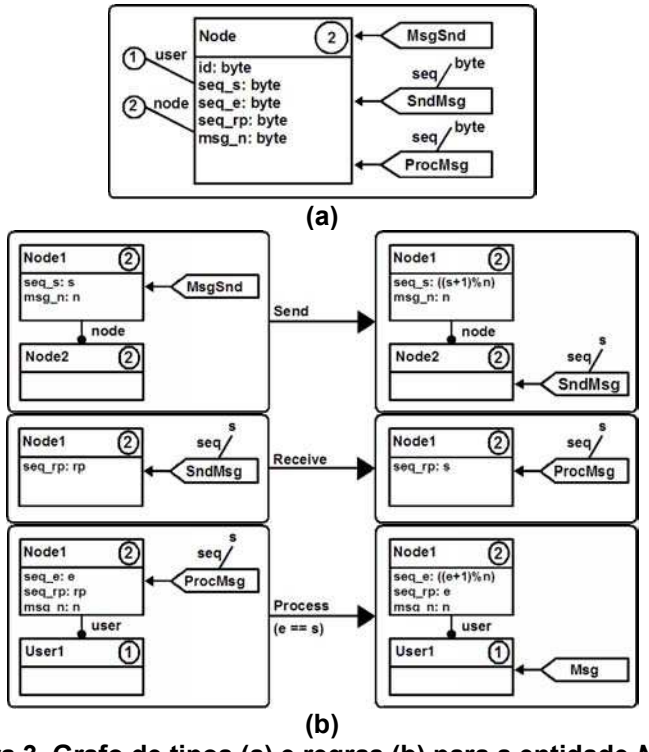

Figura 3. Grafo de tipos (a) e regras (b) para a entidade *Node*.

Além disso, um objeto do tipo *Node* pode receber três tipos de mensagens: *MsgSnd*, *SndMsg*, e *ProcMsg*. Um objeto do tipo *Node* (Figura 3 (b)) envia mensagens através da regra *Send*. As mensagens são recebidas (de forma não-ordenada) pela regra *Receive*, sendo ordenadas (pela comparação do número de seqüência da mensagem esperada e o número de seqüência da mensagem armazenada) e enviadas para o usuário na regra *Process*.
Um cenário de verificação é apresentado na Figura 4, onde existem dois objetos do tipo *User* que ficam gerando mensagens para os objetos do tipo *Node*. Os objetos *Node1* e *Node2* implementam a ordenação de mensagens para os objetos *User1* e *User2*, respectivamente.

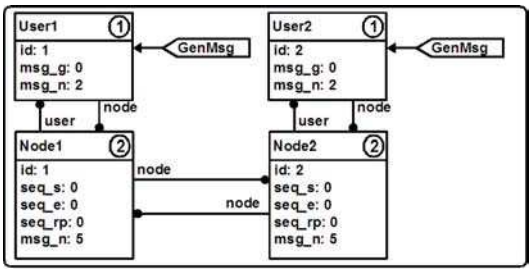

Figura 4. Grafo inicial representando um cenário de verificação.

2.2. Verificação de Modelos Descritos na GGBO

Um método para a verificação de modelos definidos na GGBO foi apresentado em [5]. No método de verificação proposto, modelos na GGBO são traduzidos para a linguagem PROMELA, a linguagem de entrada do verificador de modelos SPIN [11]. Uma importante característica da tradução de GGBO para PROMELA é a prova formal de que o modelo PROMELA traduzido mantém a semântica do modelo GGBO [5]. Em [6], o método de verificação foi ampliado para considerar a geração de contra-exemplos, de propriedades verificadas que são falsas, em termos de abstrações na GGBO.
Basicamente, no método de verificação de modelos na GGBO os usuários podem especificar propriedades na Lógica Temporal Linear (LTL), utilizada pelo SPIN, sobre a aplicação de regras do modelo GGBO sendo verificado. Ou seja, as proposições atômicas utilizadas nas fórmulas de lógica temporal devem ter como base o nome de alguma regra que compõem o modelo GGBO. Isso possibilita ao usuário especificar propriedades sobre as computações do modelo, em termos de regras aplicadas (eventos sendo gerados). Entretanto, existem muitas situações onde é necessário, para a especificação de certas propriedades, inspecionar os valores de atributos de entidades de um modelo GGBO. Em [6] é feita uma extensão do método, onde a especificação de propriedades para situações como essa são consideradas. Na Seção 3.2 é exemplificado o método de especificação e verificação de propriedades, usando a LTL em conjunto de eventos e atributos de objetos, para modelos GGBO.

3. Verificação de Sistemas Parciais

Inicialmente, nesta seção são apresentados conceitos fundamentais para a verificação de sistemas parciais, através de verificação de modelos, como os objetos distribuídos que desejamos verificar na GGBO. Na próxima seção é apresentada a metodologia utilizada para verificar sistemas parciais modelados no formalismo GGBO. A metodologia é exemplificada com o modelo de ordenação de mensagens apresentado na Seção 2.1.

Na verificação composicional a estrutura modular dos sistemas é explorada. Ou seja, cada um dos elementos que compõem o sistema são verificados de forma isolada, possibilitando que propriedades globais possam ser concluídas sobre todo o sistema [12]. Uma técnica bastante empregada na verificação composicional e difundida na literatura é a técnica *assume-guarantee* [9]. Segundo esta técnica, a especificação de um componente do sistema deve ser composta por duas partes. A primeira descreve o comportamento assumido pelo ambiente com o qual o componente interage (*assume*), e a segunda descreve o comportamento desejável do componente (*guarantee*), neste ambiente [9].

No desenvolvimento de sistemas, sistemas são diferenciados entre sistemas fechados e abertos [13]. Em um sistema fechado, o comportamento do mesmo é completamente determinado pelo estado do sistema. Um sistema aberto (ou módulo) é um sistema que interage com o ambiente e cujo comportamento depende dessa interação. Como evidenciado em [13], a verificação de modelos para sistemas abertos (módulos) deve verificar o sistema com respeito a diferentes ambientes. Mais especificamente, dado um sistema aberto (um módulo) e uma especificação de propriedade em lógica temporal, o problema de verificação de módulos consiste em verificar, para todos os ambientes possíveis, se a composição do ambiente com o módulo satisfaz a propriedade especificada em lógica temporal.

Para a verificação de módulos utilizando o raciocínio *assume-guarantee* [9], a especificação de propriedades consiste em um par $<\varphi,\psi>$. Tanto φ (*assume*, comportamento assumido do ambiente) como ψ (*guarantee*, comportamento garantido pelo módulo) são fórmulas em lógica temporal. Para a LTL, a mesma lógica temporal utilizada para a verificação de modelos descritos na GGBO [5], tanto φ como ψ são fórmulas na LTL. Quer dizer, todas as computações do componente devem garantir a especificação ψ, enquanto todas as computações do ambiente satisfizerem φ [14]. Assim, na LTL, o par $<\varphi,\psi>$ deve ser combinado em uma única fórmula LTL: $(\varphi \rightarrow \psi)$.

Porém, antes de verificar um componente do sistema isoladamente usando a técnica *assume-guarantee*, é necessário prover uma forma de completar (fechar) o comportamento do componente sendo analisado [15]. Ou seja, a modelagem do componente deve ser modificada para incluir os comportamentos necessários do ambiente, e que não fazem parte do componente, para a análise do mesmo. Como apresentado em [15], dois tipos de componentes são definidos com essa finalidade, estes são chamados de *stubs* e *drivers*.

Os *stubs* são usados para representar o comportamento de funcionalidades providas pelo ambiente, e que são necessárias para o comportamento do componente sendo verificado. Basicamente, *stubs* representam procedimentos que o componente necessita para sua execução. Todos os valores de retorno possíveis de um comportamento externo devem ser representados por *stubs*, para que o processo de verificação ocorra sobre todos os comportamentos possíveis do mesmo. Algumas restrições devem ser impostas às funcionalidades do *stub*, como, por exemplo, procedimentos representados do ambiente não podem ser recursivos [15]. Uma vez que *stubs* foram obtidos, os mesmos são inseridos dentro do componente e passam a representar o comportamento do ambiente necessário para a sua execução.

Por sua vez, os *drivers* são responsáveis pela geração de eventos de entrada do componente sendo analisado, e devem representar todas as entradas possíveis do ambiente para o componente. Mas existem casos em que não se está interessado em verificar o componente para todas as possíveis entradas do ambiente [15]. Nestes casos, o *driver* pode representar um conjunto de entradas para o componente, reduzindo a complexidade do modelo gerado para verificação.

3.1. Verificação de Sistemas Parciais modelados na GGBO

Nessa seção é apresentada a metodologia proposta para a verificação de sistemas parciais modelados na GGBO. Como discutido na seção anterior, para verificar um componente específico de um modelo é necessário completar o comportamento do componente (incluindo certos comportamentos fundamentais para o seu funcionamento, os *stubs*), e incluir eventos gerados pelo ambiente (*drivers*) para que fórmulas na LTL sejam especificadas e verificadas sobre o componente. Os passos que compõem a metodologia definida são apresentados na Figura 5.

Basicamente, um modelo na GGBO é composto por entidades (definem os atributos e mensagens que um objeto pode receber), regras (definem o comportamento dos objetos), mensagens (utilizadas para ativar as regras), e um grafo inicial (apresenta um cenário inicial para o modelo). As mensagens na GGBO são utilizadas para a ativação de regras. Nesse sentido, nós podemos classificar as mensagens em: **(i)** mensagens usadas para requisitar serviços de outras entidades, essas mensagens correspondem a interface de importação da entidade; **(ii)** mensagens utilizadas pelo ambiente para acessar a entidade, essas mensagens compõem a interface de exportação da entidade.

A classificação das mensagens que compõem uma entidade torna possível distinguir no grafo de tipos e nas regras da entidade quais serão os *stubs* e *drivers* utilizados no fechamento da entidade. Ou seja, os *stubs* serão modificações das entidades que processam as mensagens da interface de importação. Sendo que os *drivers* geram um conjunto de mensagens da interface de exportação da entidade em análise. Segundo a classificação de mensagens, uma mesma mensagem pode pertencer a mais de uma classe (interface de importação ou exportação), mas esse fato não influencia no procedimento de fechamento. Entretanto, é importante salientar que o processo de classificar se uma mensagem pertence à determinada classe ocorre, atualmente, de forma manual e cabe ao desenvolvedor realizar essa tarefa, correspondendo ao primeiro passo da metodologia.

O segundo passo da metodologia consiste em criar entidades *stub*, identificadas a partir das mensagens da interface de importação e das regras da entidade sendo verificada. As entidades *stub* apresentam a sintaxe *NomeEntidade_Stub*. No grafo de tipos elas apresentam apenas os nomes das mensagens que a entidade está apta a receber (mensagens da interface de importação da entidade sendo verificada, que acessam serviços dessa entidade *stub*), e a entidade não apresenta atributos internos. Além disto, as regras disparadas pelas mensagens definidas no grafo de tipos são modificadas. Nestas modificações, as aplicações de regras devem realizar duas possíveis operações: **(i)** consumir a mensagem que disparou a regra; **(ii)** gerar mensagens de resposta à entidade sendo analisada. No caso **(ii)**, a relação de entrada e saída do *stub* é realizada pelo desenvolvedor. Ainda no caso **(ii)**, as mensagens de resposta devem representar todas as possíveis combinações para os parâmetros que compõem essas mensagens (caso estes existirem). Apesar deste aspecto não ser tratado neste artigo, uma forma de lidar com ele consiste no desenvolvedor definir um intervalo de valores para os parâmetros das mensagens que devem ser respondidas à entidade sendo analisada. Para isso, funções de não-determinismo de geração de valores devem ser oferecidas ao desenvolvedor, para que na criação das entidades *stub* ele possa representar os valores necessários para a verificação da entidade sendo analisada.

No terceiro passo, a entidade a ser verificada é modificada de forma que as referências a outros objetos que aparecem no grafo de tipos e regras, são substituídas por entidades *stub*. Isso é facilmente resolvido, uma vez que as entidades *stub* já foram definidas e a entidade sendo analisada passa por um simples processo de substituição de referências. Após essa modificação da entidade sendo analisada, o quarto passo da metodologia consiste em

definir um *driver*. O *driver* na GGBO é um gerador de mensagens, que fazem parte da interface de exportação do objeto sendo analisado, do ambiente para o objeto. Assim, um *driver* pode ser modelado tanto como uma entidade, instanciada no grafo inicial, que fica gerando mensagens, ou como um conjunto de mensagens pré-definido (no grafo inicial) e endereçado ao objeto. Este quarto passo da metodologia é realizado manualmente pelo usuário, de acordo com o comportamento que se quer provar sobre o objeto. Logo, o usuário pode tanto definir um objeto para representar as entradas do ambiente, como gerar um conjunto de mensagens.

Após a definição do grafo inicial, a última etapa da metodologia consiste na especificação e verificação de propriedades sobre o objeto. Para isso, o usuário define fórmulas na LTL seguindo o raciocínio *assume-guarantee* ($\varphi \rightarrow \psi$), onde φ representa o comportamento esperado do ambiente e ψ o comportamento desejado do objeto. Na Figura 5, os passos descritos nessa seção, necessários para completar um objeto, especificar, e verificar propriedades sobre o mesmo são apresentados de forma resumida. Na próxima seção, a metodologia aqui discutida é exemplificada com a verificação da entidade *Node* modelada na Seção 2.1.

1) Classificar as mensagens da entidade a ser verificada em:
   a) Mensagens da interface de importação;
   b) Mensagens da interface de exportação.

2) Criar entidades que representam stubs:
   a) A partir das mensagens da interface de importação descobrir quais as entidades referenciadas e para cada uma destas entidades criar uma nova entidade que conserva o nome dela mais o sufixo "_Stub";
   b) Os grafos de tipos destas novas entidades não devem possuir atributos, mas devem conter as mensagens utilizadas para acessar os serviços da entidade stub;
   c) Incluir as regras que tratam das mensagens utilizadas para acessar os serviços da entidade stub;
   d) Modificar essas regras, de forma que atributos não são mais utilizados e a aplicação das regras apenas consome as mensagens que a disparam ou, dependendo das regras da entidade representada pelo stub, gerar mensagens de resposta à entidade a ser verificada.

3) Modificar a entidade a ser verificada:
   a) Para cada atributo que é referência, tanto no grafo de tipos como nas regras, modificar os mesmos para referenciar os stubs destas entidades.

4) Definir o driver:
   a) O usuário pode especificar um objeto que gera mensagens para a interface de exportação do objeto a ser verificado, ou definir um conjunto de mensagens para representar o comportamento a ser verificado no objeto. Ambas as formas resultam na definição de um grafo inicial.

5) Especificar propriedades sobre o comportamento do ambiente (assume) e o comportamento da entidade sendo verificada (guarantee) usando fórmulas na LTL.

Figura 5. Metodologia para a verificação de sistemas parciais modelados na GGBO.

3.2. Exemplo

Com o objetivo de exemplificar a metodologia de verificação apresentada na seção anterior, nesta seção a entidade *Node*, que implementa a ordenação de mensagens, modelada na Seção 2.1 é transformada (através do processo de fechamento) e verificada. Seguindo a metodologia definida, a partir das regras e do grafo de tipos da entidade *Node*, as mensagens enviadas ou recebidas pela entidade são classificadas em:

- Mensagens da interface de importação: *Msg* e *SndMsg*;
- Mensagens da interface de exportação: *MsgSnd*, *SndMsg*, e *ProcMsg*.

A partir dessa classificação, aplicando o segundo passo da metodologia apresentada na Figura 5, as entidades *User_Stub* e *Node_Stub* (vide Figuras 6 e 7) foram geradas. A entidade *User_Stub* não apresenta atributos, e pode receber apenas um tipo de mensagem (Figura 6 (a)): *Msg*. Como essa entidade é definida para completar o comportamento de uma entidade *Node*, ela é composta por apenas duas regras, que são disparadas pela mensagem *Msg* (Figura 6 (b)): *Consume1* e *Consume2*. Diferentemente das regras originais para a entidade *User* (vide Figura 2 (b)), essas regras foram modificadas de forma que as mensagens são apenas consumidas.
Assim como para a entidade *User_Stub*, a entidade *Node_Stub* também não apresenta atributos, podendo receber uma única mensagem (Figura 7 (a)): *SndMsg*. Essa entidade é composta pela regra *Receive*, disparada pela mensagem *SndMsg* (ver Figura 7 (b)). Essa regra foi modificada de forma que ela consome a mensagem e não realiza nenhuma operação.

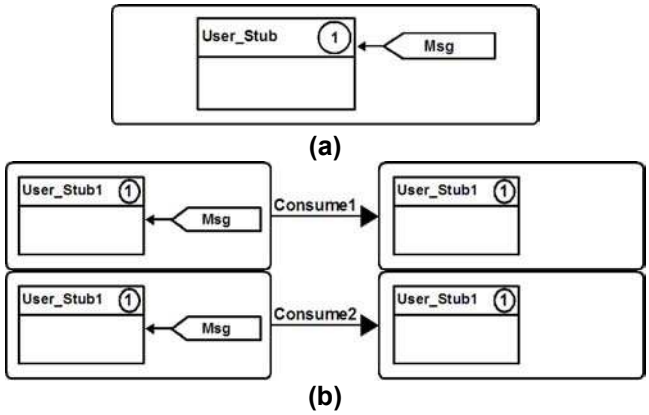

Figura 6. Grafo de tipos (a) e regras (b) para a entidade *User_Stub*.

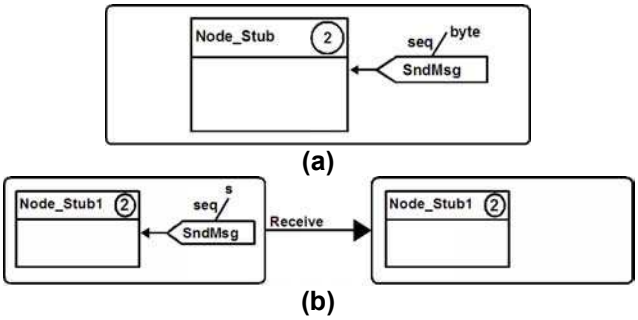

Figura 7. Grafo de tipos (a) e regras (b) para a entidade *Node_Stub*.

Nota-se, mesmo com este exemplo simples, a redução que ocorre no modelo como um todo, impactando sensivelmente no espaço de estados. Após a definição destas entidades *stub*, passamos para o terceiro passo da metodologia, onde modificamos a entidade *Node* que será verificada, substituindo as referências das entidades *User* e *Node*, respectivamente, por referências as entidades *User_Stub* e *Node_Stub*. Estas modificações podem ser visualizadas na Figura 8.
Seguindo a metodologia, o quarto passo é composto pela definição do *driver* utilizado para acessar a entidade *Node*. Neste exemplo, nós não definimos uma entidade para representar o *driver*, mas definimos mensagens, no grafo inicial, endereçadas ao objeto a ser verificado. Na Figura 9, é apresentado esse grafo inicial, composto por um objeto *Node1* que tem quatro mensagens, com números de seqüência 0, 1, 2, e 3, endereçadas ao objeto. É neste cenário que iremos verificar, utilizando o raciocínio *assume-guarantee*, se uma

instância da entidade *Node*, ordena corretamente mensagens recebidas de forma desordenada.

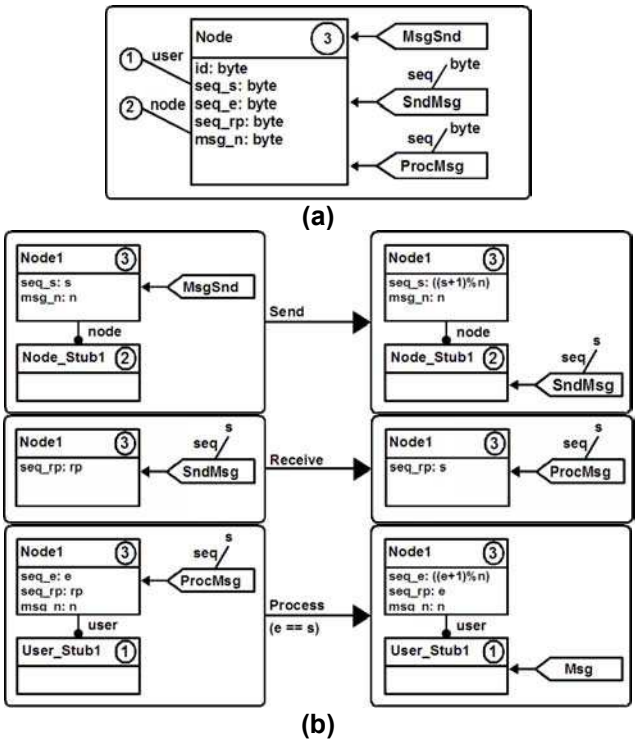

Figura 8. Grafo de tipos (a) e regras (b) para a entidade *Node* modificada.

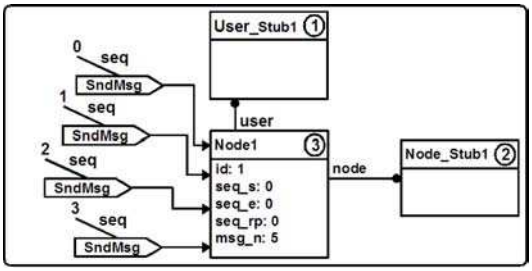

Figura 9. Grafo inicial representando um cenário para verificação de um sistema parcial.

Conforme exposto na Seção 2, a recepção de mensagens na GGBO ocorre de forma não-determinística, sendo essa característica utilizada para representar a desordenação de mensagens no exemplo. Utilizando-se de fórmulas em LTL, inicialmente nós provamos que mensagens podem chegar tanto de forma ordenada como desordenada. Para verificar essas propriedades, sobre o grafo inicial da Figura 9, nós tornamos visível o atributo *seq_rp* da entidade *Node*. Sobre o valor deste atributo e os nomes das regras da entidade *Node*, nós definimos eventos de recepção de mensagens para os vários números de seqüência. Assim, para definir a recepção da mensagem 0 (no grafo inicial definido), pela regra *Receive*, nós definimos a proposição atômica *r0*. Essa proposição trata-se de um evento em que o valor do atributo *seq_rp* é igual a 0 quando a regra *Receive* é executada no objeto.

A partir desta proposição atômica, a recepção da mensagem de número de seqüência 0 é dada pelo evento *(! r0 && X r0)*[1], seguindo a definição de eventos encontrada em [16], que

[1] A notação utilizada para os operadores temporais e booleanos das fórmulas é a mesma encontrada no verificador de modelos SPIN [11].

chamamos *R0*. Analogamente, nós definimos os eventos *R1* e *R2*. Utilizando os padrões de fórmulas LTL proposto em [16], temos padrões para expressar a presença ou ausência de um determinado evento, assim como a ordem dos mesmos. No nosso caso, para que as mensagens cheguem de forma ordenada, nós podemos provar que entre o evento *R0* e *R2* acontece o evento *R1*. Isto é dado pela fórmula:

([] ((R0 && <> R2) -> (! (! R1 U R2) && ! R2)))

Na verificação essa fórmula resultou em falsidade, gerando um contra-exemplo em que as mensagens chegam de forma desordenada no objeto. Para provar o contrário, i.e., que mensagens chegam desordenadas no objeto, nós negamos a fórmula anterior:

(! ([] ((R0 && <> R2) -> (! (! R1 U R2) && ! R2))))

Quando verificada essa fórmula também resultou em falsidade, gerando um contra-exemplo no qual mensagens chegam de forma ordenada no objeto. A partir dos contra-exemplos gerados por essas fórmulas, nós concluímos que realmente mensagens são consumidas de forma não-determinística pelo objeto, representando na verificação todos os conjuntos possíveis para as ordens das mensagens.

Para especificar que mensagens são processadas de forma ordenada, nós geramos as proposições *p0*, *p1*, e *p2*. A proposição *p0* consiste num estado em que o valor do atributo *seq_rp* é igual a 0 quando a regra *Process* é executada no objeto.

Sobre essa proposição é definido o evento *P0*. Analogamente, os eventos *P1* e *P2* são definidos, quando o valor do atributo *seq_rp* é igual a 1 e 2. Utilizando o raciocínio *assume-guarantee*, nós especificamos como fórmula *assume*:

(<> R0 && <> R1 && <> R2)

Ou seja, os caminhos de interesse são aqueles em que o ambiente gera mensagens com os identificadores 0, 1 e 2, independente da ordem da recepção das mensagens. Como comportamento garantido pelo objeto (*guarantee*), nós temos a especificação da propriedade de ordenação:

([] ((P0 && <> P2) -> (! (! P1 U P2) && ! P2)))

Logo, nós compomos as duas propriedades em uma única fórmula seguindo a técnica *assume-guarantee*:

((<> R0 && <> R1 && <> R2) -> ([] ((P0 && <> P2) -> (! (! P1 U P2) && ! P2))))

Quando verificada essa fórmula foi avaliada como verdade. Isso comprova que a seqüência de processamento de mensagens para o usuário ocorre de forma ordenada, e é respeitada em todas computações possíveis do objeto em verificação.

4. Trabalhos Relacionados

Como trabalhos relacionados ao desenvolvido no decorrer deste artigo, encontramos inúmeras contribuições objetivando a verificação de sistemas parciais [7,8,15,17,18] e verificação de módulos [13]. Além destes trabalhos relacionados, também encontramos trabalhos ligados a verificação composicional [19] e *Compositional Reachability Analysis*

(CRA) [20,21,22,23] que tratam do problema da análise composicional de um sistema, i.e., encontrar propriedades globais do sistema a partir de propriedades locais dos seus subsistemas.

Uma abordagem automática para completar sistemas parciais reativos abertos é descrita em [8]. Esta abordagem objetiva a produção de sistemas fechados capazes de serem executados no contexto do *toolset* Verisoft [24]. O complemento gerado para o sistema age como um ambiente de controle que leva à exploração automática do comportamento do sistema parcial em estudo. Para produzir um complemento tratável, é realizado um conjunto de análises para determinar quais porções do sistema parcial podem ser influenciadas pelo comportamento externo. Ainda, além dos trabalhos referenciados em [15,17], em [18] são apresentadas técnicas que permitem sistemas parciais serem descritos como uma mistura de código fonte e especificação. Nestes trabalhos, especificações podem ser tomadas como assunções de um sistema parcial dado em código, objetivando a análise automática de propriedades de temporização do sistema.

A abordagem de [19] comporta tanto a verificação de componentes a partir de assunções do ambiente e sem um contexto de composição específico, quanto a composição de módulos verificados para construção de um sistema. Propriedades de um módulo primitivo são obtidas diretamente a partir da verificação de sua representação original, a qual pode ser constituída através de qualquer linguagem sujeita a aplicação de *model-checking*, como subconjuntos verificáveis de UML, Java ou C/C++. Os módulos primitivos tipicamente apresentam-se como sistemas abertos, sendo realizado o fechamento e verificação do módulo de maneira semelhante à utilização de *stubs* e *drivers*. O conjunto de propriedades verificadas para um módulo primitivo passa a constituir uma abstração desse elemento. Tal abstração é utilizada em etapas de construção de módulos compostos a partir de primitivos. Para a composição é realizada uma análise sobre as propriedades verificadas para os módulos sendo compostos em cada passo, visando determinar se estas são válidas na composição.

Abordagens usando o paradigma *assume-guarantee* dependem de alguma interação humana para a determinação de um conjunto adequado de assunções para cada situação. Tal fato limita a possibilidade de realização de uma verificação de forma completamente automática. Na tentativa de contornar tal necessidade existem pesquisas em métodos para geração automática de assunções em raciocínio composicional, como [25]. Em [25] é proposta uma abordagem para automatizar a geração de assunções para propriedades de *safety* no contexto de representações de sistemas e seus componentes usando *Labeled Transitions Systems* (LTSs). Nessa abordagem, uma assunção gerada automaticamente pode ser tão complexa e de verificação tão custosa quanto outros componentes.

O uso de assunções para sintetizar um modelo do ambiente feito em [7] é similar ao trabalho realizado em CRA de [20,21,22,23]. Essas abordagens decompõem um sistema em subsistemas, os quais correspondem a módulos. Os módulos identificados são representados por LTSs ou representações composicionais similares a grafos de transição. Um LTS de mais alto nível é minimizado de acordo com as propriedades sendo checadas e demais restrições de contexto de forma que sejam obtidas interfaces que resumem o comportamento de cada módulo, para então realizar as análises utilizando as interfaces obtidas em lugar dos detalhes dos módulos, diminuindo assim o espaço de estados gerados para a verificação. CRA aplica-se a um sistema fechado ou a seus módulos no contexto do sistema completo. Essa noção de capturar o comportamento do ambiente através de interfaces também aparece em resultados teóricos relativos à verificação modular como em [26,27,28].

Artigos como [15,7,17,18,8] apresentam metodologias para proceder quanto a verificação de sistemas abertos. [19], por sua vez, apresenta uma abordagem para a composição dos

resultados obtidos a partir da análise desses módulos. Abordagens modulares são úteis por possibilitarem tanto a verificação de sistemas parciais quanto por oferecerem uma maneira de contornar o problema de explosão de estados para a verificação de sistemas maiores. A abordagem apresentada nesse artigo não introduz novidades em respeito à análise de módulos ou sistemas abertos, mas sim adapta elementos de metodologias dessa área para a aplicação utilizando-se modelos em GGBO. Em especial, as noções de fechamento (construção de *stubs* e *drivers*) e verificação usando a técnica *assume-guarantee* (através da LTL).

5. Considerações Finais

Neste artigo foi apresentada uma metodologia para a verificação de sistemas parciais modelados na Gramática de Grafos Baseada em Objetos (GGBO). Um sistema descrito por um modelo baseado em objetos apresenta uma arquitetura simples (devido à abstração) e descentralizada (devido à sua estrutura de objetos), facilitando a aplicação da técnica *assume-guarantee*, especialmente quando os objetos do sistema são reativos (como ocorre na GGBO).

O trabalho desenvolvido soma-se aos trabalhos já existentes para a GGBO, especialmente na área de verificação de modelos [5,6]. Mais especificamente, o trabalho apresentado possibilita que objetos específicos de um modelo sejam verificados de tal forma que os estados gerados na verificação sejam muito menores, facilitando a verificação de modelos maiores.

Como pode ser visto no desenvolver do artigo, o principal procedimento por trás da metodologia de verificação, consiste no procedimento de fechamento. Nesse sentido, o procedimento apresentado no artigo não ocorre atualmente de forma automática, visto que inúmeros pontos devem ser tratados de forma específica pelo desenvolvedor que está modelando o sistema. Entretanto, a automatização do processo representa um importante trabalho futuro. Outro trabalho futuro consiste na definição de funções, para a geração não-determinística de valores, que possibilitem ao desenvolvedor verificar seus objetos num cenário em que todas as computações possíveis do ambiente são representadas. Nesse sentido, a adição dessas computações podem gerar a explosão de estados do modelo, e seu uso também deve ser melhor avaliado.

Além destes, outro trabalho futuro consiste na integração das provas realizadas de forma separada (utilizando a metodologia aqui proposta), sobre os objetos de um modelo, para que provas globais de um modelo (contendo vários tipos de objetos já verificados) sejam feitas. Ou seja, realizar uma análise composicional sobre o sistema. Essa análise composicional tem como objetivo reduzir a possibilidade de explosão de estados para verificação de modelos na GGBO. Para tanto, técnicas para a verificação composicional devem ser utilizadas ou definidas.

Referências

1. Silva, F. A. A transaction model based on mobile agents. Tese de doutorado, Technical University Berlin - FB Informatik, Alemanha, 1999.
2. Dotti, F. L., Ribeiro, L. Specification of mobile code systems using graph grammars. In: 4th International Conference on Formal Methods for Open Object-Based Distributed Systems, volume 177, IFIP Conference Proceedings, p. 45-63, EUA, 2000. Kluwer Academic Publishers.
3. Copstein, B., Móra, M. C., Ribeiro, L. An environment for formal modeling and simulation of control systems. In: 33rd Annual Simulation Symposium, p. 74-82, EUA, 2000. IEEE Computer Society Press.

4. Dotti, F. L., Duarte, L. M., Copstein, B., Ribeiro, L. Simulation of mobile applications. In: 2002 Communication Networks and Distributed Systems Modeling and Simulation Conference, p. 261-267, EUA, 2002. The Society for Modeling and Simulation International.
5. Dotti, F. L., Foss, L., Ribeiro, L., Santos, O. M. Verification of object-based distributed systems. In: 6th International Conference on Formal Methods for Open Object-Based Distributed Systems, volume 2884, LNCS, p. 261-275, França, 2003. Springer-Verlag.
6. Santos, O. M., Dotti, F. L., Ribeiro, L. Verifying object-based graph grammars (aceito para publicação). ENTCS, p. 1-12, 2004.
7. Pasareanu, C. S., Dwyer, M. B., and Huth, M. Assume-guarantee model checking of software: a comparative case study. In: 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, volume 1680, LNCS, p. 168-183, França, 1999. Springer-Verlag.
8. Colby, C., Godefroid, P., Jagadeesan, L. J. Automatically closing open reactive programs. ACM SIGPLAN Notices, 33(5):345-357, 1998.
9. Pnueli, A. In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems, NATO ASI F13, p. 123-144, EUA, 1985. Springer-Verlag.
10. Ehrig, H. Introduction to the algebraic theory of graph grammars. In: 1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology, volume 73, LNCS, p. 1-69, Alemanha, 1979. Springer-Verlag.
11. Holzmann, G. J. The model checker SPIN. IEEE Transactions on Software Engineering, 23(5):279-295, 1997.
12. Clarke, E. M., Grumberg, O., Peled, D. A. Model checking. MIT Press, EUA, 1999.
13. Kupferman, O., Vardi, M. Y., Wolper, P. Module checking. Information and Computation, 164(2):322-344, 2001.
14. Vardi, M. Y. Branching vs linear time: final showdown. In: 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, volume 2031, LNCS, p. 1-22, Itália, 2001. Springer-Verlag.
15. Dwyer, M. B., Pasareanu, C. S. Filter-based model checking of partial systems. In: 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, p. 189-202, EUA, 1998. ACM Press.
16. Paun, D. O., Chechik, M. Events in linear-time properties. In: 4th International Conference on Requirements Engineering, p. 123-132, Irlanda, 1999. IEEE Computer Society Press.
17. Dwyer, M. B., Pasareanu, C. S. Model checking generic container implementations. In: Selected Papers from the International Seminar on Generic Programming, volume 1766, LNCS, p. 162-177, Reino Unido, 2000. Springer-Verlag.
18. Avrunin, G. S., Corbett, J. C., Dillon, L. K. Analyzing partially-implemented real-time systems. IEEE Transactions on Software Engineering, 24(8):602-614, 1998.
19. Xie, F., Browne, J. C. Verified systems by composition from verified components. In: 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering, p. 277-286, Finlândia, 2003. ACM Press.
20. Graf, S., Steffen, B. Compositional minimization of finite state systems. In: 2nd International Workshop on Computer Aided Verification, volume 531, LNCS, p. 186-196. Springer-Verlag, 1991.
21. Yeh, W. J., Young, M. Compositional reachability analysis using process algebra. In: Symposium on Testing, Analysis, and Verification, p. 49-59, Canadá, 1991. ACM Press.

22. Cheung, S. C., Kramer, J. Context constraints for compositional reachability analysis. ACM Transactions Software Engineering Methodology, 5(4):334-377, 1996.
23. Cheung, S. C., Kramer, J. Checking safety properties using compositional reachability analysis. ACM Transactions Software Engineering Methodology, 8(1):49-78, 1999.
24. Godefroid, P. Model checking for programming languages using VeriSoft. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 174-186, França, 1997. ACM Press.
25. Giannakopoulou, D., Pasareanu, C. S., Barringer, H. Assumption generation for software component verification. In: 17 th IEEE International Conference on Automated Software Engineering, p. 3-12, Reino Unido, 2002. IEEE Computer Society.
26. Manna, Z., Pnueli, A. The temporal logic of reactive and concurrent systems - specification. Springer-Verlag, Alemanha, 1992.
27. Kupferman, O., Vardi, M. Y. On the complexity of branching modular model checking (extended abstract). In: 6th International Conference on Concurrency Theory, volume 962, LNCS, p. 408-422, EUA, 1995. Springer-Verlag.
28. Kupferman, O., Vardi, M. Y. An automata-theoretic approach to modular model checking. ACM Transactions on Programming Languages Systems, 22(1):87-128, 2000.