Detecção Automática de Estilos Arquiteturais em Sistemas Distribuídos

Aline S. do A. Divino e Sandro S. Andrade
Especialização em Computação Distribuída e Ubíqua
Grupo de Pesquisa em Sistemas Distribuídos, Otimização, Redes e Tempo-Real
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Salvador, Bahia
Email: {alinedivino, sandroandrade}@ifba.edu.br

***Resumo*—Os sistemas de *software* vêm se tornando parte essencial da vida na sociedade moderna e nas grandes organizações. A exigência de prazos rigorosos e melhor qualidade têm feito uma disciplina ganhar importância: a arquitetura de *software*. A análise arquitetural tem como objetivo a definição de métodos e ferramentas para avaliação de qualidade em arquiteturas de *software*. Este trabalho apresenta um mecanismo para detecção automatizada de estilos arquiteturais em sistemas distribuídos. A proposta utiliza técnicas de *model checking* baseadas em gramática de grafos e implementa uma funcionalidade para detecção de estilos arquiteturais utilizando as técnicas mencionadas para efetuar a detecção. A funcionalidade é implementada como um novo recurso no GROOVE (*GRaphs for Object-Oriented Verification*), ferramenta *open source* destinada à simulação e análise de transformações em grafos, bastante versátil. No trabalho são apresentados os benefícios inferidos pela automatização da análise arquitetural – como por exemplo, a detecção de desvios ou erosões arquiteturais – de modo a permitir melhor compreensão da arquitetura.**

Palavras chaves - arquitetura de *software*; estilos arquiteturais; sistemas distribuídos; análise arquitetural; gramáticas de grafos; *model checking*; GROOVE.

I. Introdução

A arquitetura de *software* é o conjunto das principais decisões sobre a construção de sistema [1]. Ela é formada por uma coleção de: elementos para processamento, armazenamento de dados e conexão; propriedades e relacionamentos que definem uma forma arquitetural; e descrição das intenções, premissas e restrições que justificam a arquitetura (*rationale*). As principais decisões de projeto abrangem muitos aspectos do sistema em desenvolvimento, tais como: a forma como o sistema deve estar estruturado, a forma de comunicação entre os seus componentes, requisitos de implementação como por exemplo, necessidade de implementar determinadas *interfaces*, dentre outros aspectos. Estas decisões são objeto de estudo e investigação da atividade de análise arquitetural, que permite a compreensão da arquitetura do *software* através de diferentes perspectivas e níveis de abstração [2].

Segundo Taylor et al. [2], a arquitetura de *software* é essencial à qualidade de um sistema. Ela está associada a um conjunto de requisitos não-funcionais (ou atributos de qualidade) que o sistema deve atender [2]. Dentre esses requisitos pode-se citar: eficiência, confiabilidade, integridade, usabilidade, flexibilidade, segurança, portabilidade, manutenibilidade e vários outros requisitos que podem estar presentes na arquitetura a depender da natureza do sistema. Nos sistemas atuais, os requisitos não-funcionais a serem atendidos são tão importantes quanto fazer com que o *software* responda corretamente [3].

No projeto arquitetural de sistemas distribuídos essa necessidade é ainda maior, devido à natureza desse tipo de sistema (ex: processamento paralelo e descentralizado) [2], [3]. Sistemas distribuídos são formados por componentes independentes que executam em dispositivos distintos, ligados por uma rede de computadores. A forma na qual tais componentes encontram-se distribuídos – bem como os mecanismos de comunicação entre componentes – são determinantes para o desempenho e eficiência do sistema [3].

Segundo Baier [4], a sociedade está cada vez mais dependente de computadores e sistemas dedicados a ajudar em quase todos os aspectos da vida diária. As soluções baseadas em *software* estão incorporadas ao cotidiano em quase todas as áreas de atividades no mundo, tornando-se uma parte essencial da vida na sociedade moderna e grandes organizações. Quanto mais críticas e indispensáveis, mais complexa e cara se torna a concepção e construção de tais aplicações [2]. Dentro deste contexto, projetar uma boa arquitetura de *software* representa um dos fatores determinantes para o sucesso do sistema [2]. A arquitetura de *software* projetada para a construção do sistema é chamada de arquitetura prescritiva. Ela é um conjunto formado pelas principais decisões arquiteturais tomadas pelos arquitetos em um tempo qualquer. É a prescrição para construção do sistema [1]. A arquitetura descritiva é composta pelas principais decisões arquiteturais que foram de fato implementadas [1].

Ao longo da vida de um sistema, várias revisões da arquitetura prescritiva e descritiva são realizadas. Em um cenário ideal, essas arquiteturas deveriam ser idênticas, porém isso nem sempre acontece [2]. As discrepâncias existentes entre as arquiteturas prescritiva e descritiva do sistema são responsáveis pela degradação arquitetural do *software*. Segundo Taylor et al. [2], há dois tipos fundamentais de degradação arquitetural:

- Desvio arquitetural: ocorre quando há a introdução, na arquitetura descritiva do sistema, de decisões principais de projeto que: *i*) não estão incluídas na arquitetura prescritiva ou não são implicações dela, mas *ii*) não violam nenhuma das decisões de projeto da arquitetura prescritiva;
- Erosão arquitetural: é a introdução, na arquitetura

descritiva do sistema, de decisões principais de projeto que violam decisões da arquitetura prescritiva.

A ocorrência de desvios arquiteturais refletem a negligência do engenheiro em relação à conformidade do sistema com a arquitetura prescritiva. Situações como estas podem ocasionar perdas na compreensão do sistema, uma vez que importantes decisões de projeto estão implementadas, porém não estão descritas na arquitetura prescritiva. Os desvios arquiteturais, quando não corrigidos, frequentemente evoluem para erosões arquiteturais [2]. O processo de erosão arquitetural faz com que os benefícios proporcionados por um bom projeto arquitetural sejam anulados. Segundo Perry et al. [1], as frequentes violações produzem sistemas difíceis de entender e adaptar, além de contribuir para o aumento de potenciais falhas no *software*. Durante a evolução do *software* a ocorrência de desvios e erosões arquiteturais pode acontecer por vários motivos: *i*) prazos curtos que impedem a análise e documentação do impacto, da implementação na arquitetura prescritiva e *ii*) ausência de documentação das principais de decisões de projeto da arquitetura prescritiva. Em situações como estas, a atividade de análise arquitetural assume um papel de grande relevância, ajudando a verificar ocorrência de desvios e erosões durante desenvolvimento do *software* [2].

Uma das abordagens propostas para a representação arquitetural é o uso de estilos arquiteturais [2]. Eles definem entre outros aspectos, os tipos de componentes e conectores utilizados em uma descrição arquitetural [3]. No projeto arquitetural de *software* uma das principais decisões se refere à escolha do estilo arquitetural a ser seguido. O arquiteto de *software* pode decidir por adotar uma coleção de estilos arquiteturais para satisfazer certos requisitos não-funcionais [2]. Um estilo arquitetural é uma especificação de tipos de elementos e relações, com um conjunto de restrições sobre como eles podem ser usados pelo sistema [3].

Uma das grandes vantagens no uso de estilos arquiteturais está na caracterização de uma família de sistemas em termos de seus padrões de organização estrutural e comportamental [5]. Cada estilo arquitetural privilegia um conjunto de atributos de qualidade, enquanto desfavorece outros [6]. Dificilmente uma aplicação segue um único estilo arquitetural. Segundo Shaw et al. [5] e Clements et al. [3], cada estilo arquitetural é adequado a uma classe de problemas mas nenhum é adequado para solucionar todos os problemas, por isso normalmente os estilos são usados de forma combinada. Segundo ChenLi et al. [7], um estilo arquitetural pode ser representado por gramáticas de grafos, possibilitando a verificação do estilo e suas transformações ao longo da evolução da arquitetura do *software*.

Gramáticas de grafos são notações formais que ajudam a modelar e representar o *software* [7]. Nas gramáticas de grafos, os elementos básicos da arquitetura do *software* – componentes e conectores – podem ser mapeados em vértices, e suas ligações em arestas. Dessa forma, as gramáticas de grafos definem precisamente o sistema como um grafo e capturam mudanças na definição da arquitetura [7]. Notações formais, como gramáticas de grafos, têm sido bastante usadas na representação arquitetural de aplicações complexas. Elas permitem representar a arquitetura do *software* com rigor matemático e pode ser usada em conjunto com técnicas de *model checking* (verificação de modelos). Segundo Baier et al. [4], as técnicas de *model checking* permitem verificar de forma automatizada se determinadas propriedades estão presentes no sistema.

Este trabalho apresenta o projeto e implementação de um mecanismo para automatizar a detecção de estilos arquiteturais presentes em sistemas distribuídos. O mecanismo tem como meta verificar se uma dada arquitetura implementa determinado estilo arquitetural específico para sistemas distribuídos. Para isso foi implementada uma ferramenta de detecção de estilos arquiteturais presentes em um sistema, com base em gramáticas de grafos e técnicas de *model checking*. A ferramenta foi construída como um novo módulo do GROOVE – ferramenta *open source* para verificação de modelos e transformações em grafos. Dessa forma, foi possível utilizar os recursos de manipulação de gramáticas de grafos e *model checking* que o GROOVE já dá suporte e implementar um novo tipo de verificação. O restante deste trabalho está organizado como segue. A Seção II descreve os principais estilos arquiteturais para sistemas distribuídos utilizados neste trabalho. A Seção III apresenta o problema investigado neste trabalho e as técnicas utilizadas para tratar o problema. A Seção IV descreve o mecanismo de detecção proposto e seu funcionamento. A Seção V apresenta a avaliação do mecanismo para detecção de estilos arquiteturais em sistemas distribuídos. A Seção VI cita os principais trabalhos correlatos. A Seção VII apresenta as limitações do trabalho e a Seção VIII apresenta as conclusões e sugestões de trabalhos futuros.

II. Estilos Arquiteturais para Sistemas Distribuídos

Segundo Taylor et al. [2], o termo sistema distribuído é usado para designar sistemas computacionais que estão distribuídos em múltiplos processos, localizados em computadores conectados através de uma rede de computadores e executando de forma integrada através de troca de mensagens. O desenvolvimento de sistemas distribuídos é uma atividade nem um pouco trivial, pois envolve tanto o atendimento dos requisitos não-funcionais, bem como dos requisitos da aplicação, como protocolos de comunicação e gerenciamento de transações distribuídas [3]. Adotar métodos que atestem a aderência do *software* aos requisitos funcionais e não-funcionais esperados é importante para garantir um produto final com a qualidade desejada.

Um importante aspecto considerado na arquitetura de sistemas distribuídos está no uso de estilos arquiteturais para induzir ao sistema os atributos de qualidade (ou requisitos não-funcionais) desejados, além de proporcionar menos esforço para entender a arquitetura do *software*. Existem catalogados diversos estilos arquiteturais, para os quais são citados benefícios e contrapartidas na sua escolha como direcionadores para o projeto de uma arquitetura de *software* [3]. Segundo Shaw et al. [5], os estilos catalogados foram definidos observando-se a arquitetura de sistemas existentes. Percebeu-se, com base em estudos de casos, que os sistemas estavam organizados de forma similar e faziam uso de estruturas semelhantes [5]. Dentre os estilos arquiteturais catalogados para sistemas distribuídos [3], pode-se destacar os estilos descritos a seguir:

A. Client-Server

É um tipo particular de *Virtual Machine* (arquitetura em camadas) na qual a aplicação é organizada em duas camadas: camada servidor, responsável por todo o processamento da aplicação e o gerenciamento de dados e camada cliente, responsável por executar o *software* de apresentação [2]. Os componentes interagem solicitando serviços de outros componentes. Os solicitantes são denominados clientes, e os fornecedores de serviços são os servidores, que fornecem um conjunto de serviços por meio de uma ou mais portas [3]. Cada servidor possui em sua porta a descrição do serviço que ele fornece e cada cliente possui portas que descrevem os serviços que ele precisa solicitar. Toda comunicação nesse tipo de arquitetura deve ser sempre iniciada pelos clientes, ou seja, nunca um servidor irá solicitar algum serviço a um componente cliente. O principal tipo de conector nesse tipo de arquitetura é o *remote procedure calls* – protocolo de comunicação entre processos que permite solicitar serviços de outros processos localizados geralmente em outro computador, conectado por uma rede, sem a necessidade de entender os detalhes da rede [3].

Nesse tipo de arquitetura pode haver servidores centrais ou múltiplos servidores distribuídos [3]. Um exemplo típico de sistema com estilo *Client-Server* são as aplicações Web: os clientes rodam no navegador Web e o servidor roda em um servidor Web, como o *Weblogic*. Dentre as qualidades induzidas por esse estilo pode-se destacar:

- Centraliza a computação e os dados no servidor, tornando a informação disponível para os múltiplos clientes. Nessa situação um único servidor poderoso pode servir a muitos clientes [3];
- Simplifica os componentes e possibilita sua otimização e reutilização [3];
- Viabiliza o balanceamento de carga, uma vez que qualquer servidor que utilize o mesmo banco de dados pode tratar a requisição [3].

No modelo arquitetural do estilo, os clientes iniciam a comunicação, invocando serviços conforme sua necessidade e esperam ou ficam bloqueados até que o serviço solicitado complete sua ação, fornecendo um resultado de retorno. Assim, o cliente deve saber a identidade de serviço para invocá-lo. Em contraste, os servidores não sabem a identidade dos clientes até que uma solicitação seja feita ao servidor. A Figura 1 ilustra este cenário.

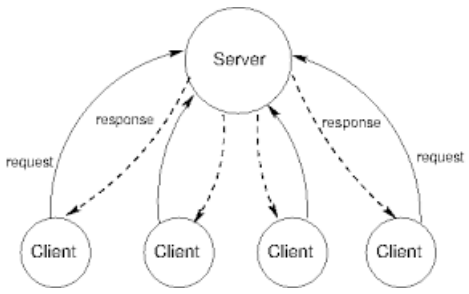

Figura 1: Funcionamento básico do estilo *Client-Server*.

Segundo Clements et al. [3], a arquitetura do estilo *Client-Server* apresenta a visão do sistema separando a aplicação cliente dos serviços por ela usados. Este estilo apoia o entendimento e reutilização do sistema e a fatoração de serviços comuns. Como os servidores podem ser acessados por qualquer número de clientes, é relativamente fácil adicionar novos clientes a um sistema. Da mesma forma, servidores podem ser replicados para suportar uma maior escalabilidade e disponibilidade.

B. Peer-To-Peer

No estilo arquitetural *Peer-To-Peer*, os componentes interagem diretamente como pares por meio de troca de serviços [3]. Cada componente é do tipo *peer* e pode produzir e consumir serviços similares. Na arquitetura *Peer-To-Peer* a comunicação é uma espécie de interação de pedido ou resposta sem a assimetria encontrada no estilo arquitetural *Client-Server*, isto é, qualquer componente pode, a princípio interagir com qualquer outro componente, solicitando os seus serviços. Os componentes no *Peer-To-Peer* são tipicamente programas independentes que funcionam em diversos nós de uma rede de computadores. Exemplos de aplicações *Peer-To-Peer* incluem redes de compartilhamento de arquivos como o *BitTorrent* [8] e aplicações de mensagem instantâneas e VoIP [9] como o *Skype* [10].

Os conectores em arquiteturas *Peer-To-Peer* podem envolver protocolos bidirecionais de interação complexa, refletindo a comunicação de duas vias que pode existir entre dois ou mais componentes no *Peer-To-Peer*. O principal tipo de conector no estilo *Peer-To-Peer* é o *Call-Return*. Ao contrário do estilo de *Client-Server*, a comunicação pode ser iniciada por qualquer uma das partes e cada *peer* possui *interfaces* que descrevem os serviços que solicita de outros *peers* e os serviços disponibilizados [3]. Dentre as qualidades induzidas por esse estilo pode-se citar: computação descentralizada, onde as computações podem ser realizadas por qualquer *peer* distribuído na rede e maior disponibilidade [3].

No estilo arquitetural *Peer-To-Peer*, os *peers* interagem diretamente entre si e podem desempenhar o papel de solicitante e fornecedor de serviços, assumindo qualquer um dos papéis para a tarefa em questão. Este compartilhamento de papéis fornece flexibilidade para implantar o sistema como uma plataforma altamente distribuída. Os *peers* podem ser adicionados e removidos na rede sem impacto significativo, o que resulta em grande escalabilidade para todo o sistema [3]. A arquitetura *Peer-To-Peer*, pode ter tipos especiais de *peers* chamados *ultrapeers*, *ultranodes* ou *supernodes*. Esses tipos de *peers* possuem capacidade de indexação ou roteamento, permitindo que um *peer* convencional consiga conhecer um maior número de outros *peers* [3]. Um exemplo de sistema *Peer-To-Peer* onde se aplica o conceito de *supernodes* é o Skype. Nesse sistema cada usuário é considerado um *peer*, que inicialmente deve se conectar ao servidor de *login* do *Skype* para recebe o IP de um *supernode* e a partir daí iniciar a comunicação *Peer-To-Peer*, conforme ilustrado na Figura 2.

C. REST (REpresentational State Transfer)

O REST (*REpresentational State Transfer*) é um estilo arquitetural voltado para produção de aplicações Web distribuídas, extensíveis e altamente escaláveis [11]. Ele consiste num

conjunto de heranças arquiteturais de outros estilos como o *Client-Server* e o *Mobile Code* (variação do *Code-on-Demand*) [2]. No REST, as restrições arquiteturais do *Client-Server* são aplicadas a componentes, conectores e elementos de dados dentro da *Web*. Ele incorpora a extensão dinâmica do *Mobile Code* ao permitir que clientes recebam dados arbitrários, descritos por meta-dados, e permitir que suas funcionalidades possam ser estendidas dinamicamente.

O foco desta arquitetura é a comunicação de recursos (elementos de dados) entre clientes e servidores de maneira única. Os recursos são identificados através de URIs (*Uniform Resource Identifiers*) e para manipular estes recursos, os componentes podem se comunicar através de *interfaces* como *sockets* ou protocolos de comunicação como o HTTP (*Hypertext Transport Protocol*). Toda a interação com um recurso deve ser mediada através de uma representação do mesmo, em que todos os componentes, convergem no entendimento da codificação daquela representação. A Figura 3 apresenta este cenário.

O estilo REST é um modelo arquitetural capaz de atender requisitos complexos de integração sem agregar complexidade e excesso de acoplamento [11]. Dentre as qualidades induzidas por esse estilo pode-se citar:

- Separação em camadas para melhorar a eficiência, possibilitar a evolução independente dos elementos do sistema e prover robustez;
- Replicação para reutilizar informação e diminuir a latência e contenção;
- Extensão dinâmica através de *mobile code*;
- Requisições ao servidor são sempre *context-free*, que geram escalabilidade e robustez.

Segundo Fielding et al. [11], o REST é um estilo arquitetural que se adapta muito bem a um grande número de clientes, permite a transferência de recursos em fluxos de tamanho ilimitado e suporta diferentes topologias como *multi-client* ou *multi-server* com *proxies* intermediários.

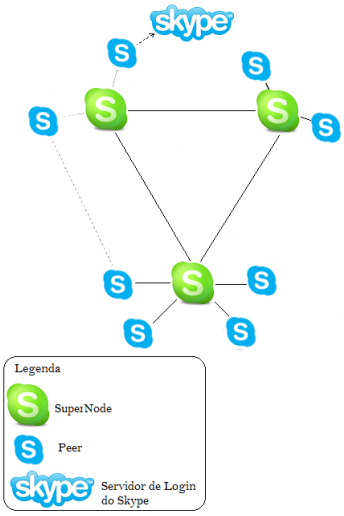

Figura 2: Skype - Exemplo de sistema *Peer-To-Peer*.

D. MapReduce

O *MapReduce* é um estilo arquitetural para computação distribuída, baseado num paradigma de programação para processamento de grandes coleções de dados, empregando o poder da computação distribuída e paralela [12]. O *MapReduce* é um modelo de programação desenvolvido pelo Google para armazenamento e processamento de grandes conjuntos de dados em ambientes distribuídos [13]. Arquiteturas que adotam este estilo são caracterizadas pela presença de dois tipos básicos de componentes: *Map* e *Reduce*. O *Map* é responsável por receber os dados organizados no formato chave/valor e fazer a carga, transformação e processamento de dados. O *Reduce* é responsável por atuar sobre os valores produzidos pelas funções de *Map*, agrupando-os e produzindo uma nova lista. O *MapReduce* desempenha um papel fundamental na computação em nuvem, uma vez que diminui a complexidade da programação distribuída e é fácil de ser desenvolvida em grandes aglomerados de máquinas comuns. As aplicações *MapReduce* são projetados para processar um grande volume de dados paralelamente. Isso exige a divisão da carga de trabalho em um grande número de máquinas. O *Apache Hadoop* oferece uma maneira sistemática de implementar este paradigma de programação [14].

O *Apache Hadoop* é um *framework* de código aberto para o armazenamento e processamento de dados em larga escala [14]. Ele oferece como principal ferramenta uma implementação do *MapReduce*, responsável pelo processamento distribuído, e o HDFS (*Hadoop Distributed File System*) para armazenamento de grande volume de dados em diversas máquinas organizadas em *clusters*. O *Apache Hadoop* tem se destacado como uma ferramenta eficaz sendo amplamente usado por muitas aplicações, tais como *FaceBook*, *Yahoo*, *Twitter*, dentre outras [13]. O funcionamento do *Apache Hadoop* é baseado num conjunto de *daemons*, serviços que executam quando o *Hadoop* é iniciado. Dentre os *daemons* fundamentais para sua execução temos:

- *NameNode*, serviço mestre que gerencia o HDFS e todos os *DataNodes*. Nas versões mais novas do *Apache Hadoop*, o *cluster* pode possuir mais de um *NameNode*, de modo a permitir que o funcionamento dos *DataNodes* não seja prejudicado caso um *NameNode* falhe;
- *DataNode*, serviço escravo que gerencia o armazenamento e processamento na máquina em que está executando;
- *ResoucerManager*, principal *daemon* e sua função é alocar e monitorar o conjunto de recursos necessários para todos as aplicações a serem executadas no sistema.

A comunicação entre os componentes no *MapReduce* é realizada por conectores do tipo *remote procedure call* ou conectores do tipo *stream*, para enviar e receber dados simultaneamente [12]. Tanto os componentes de *Map* como os de *Reduce* são implementados como tarefas que executam em um *cluster*. A entrada para uma tarefa de *MapReduce* é um conjunto de arquivos, armazenados no HDFS. Dentre as qualidades induzidas está permitir resolver as dificuldades que surgem com o processamento distribuído, como falhas

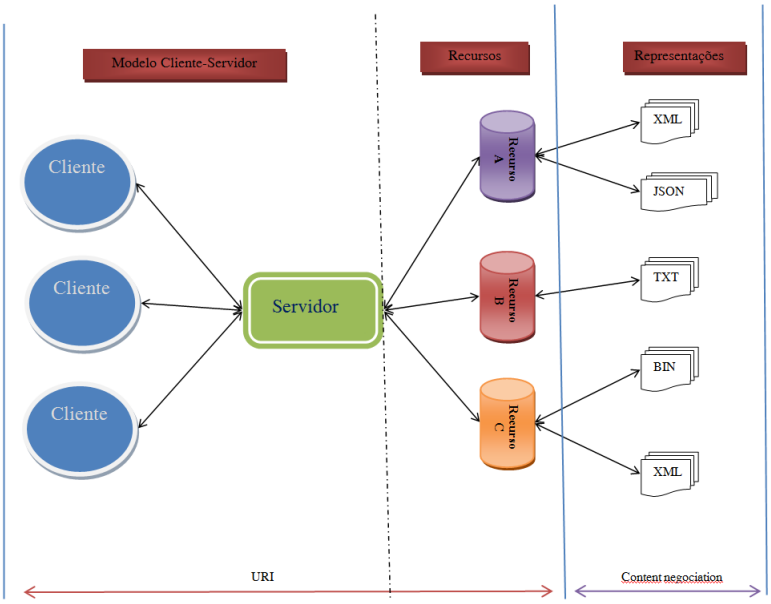

Figura 3: Exemplificando os conceitos e mecanismo relacionados ao REST [11].

decorrentes da inatividade de nós e problemas que surgem com o compartilhamento e replicação de informações que estão na rede [12]. A Figura 4 apresenta um cenário de *MapReduce*.

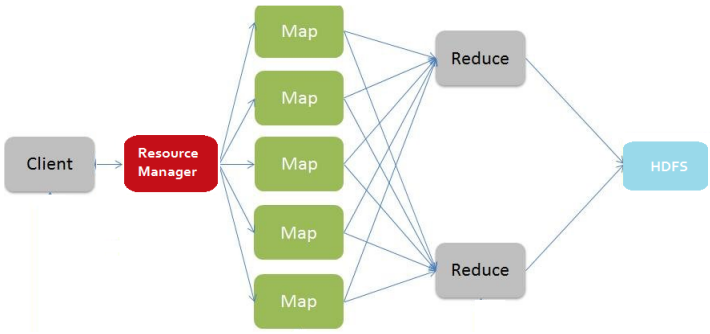

Figura 4: Visão simplificada do estilo *MapReduce*.

No estilo arquitetural *MapReduce* podem existir apenas tarefas de *Map*, é o caso do *Design Patterns* para *MapReduce*, *filtering*, usado nos casos em que o volume de dados é extremamente grande e deseja-se apenas obter um subconjunto de dados para análise. A compreensão da arquitetura do *MapReduce* ajuda na construção de aplicações críticas que trabalham com *BigData* [15].

III. Detecção de Estilos Arquiteturais em Sistemas Distribuídos

Nos últimos anos, uma série de novas demandas têm feito com que o projeto de arquiteturas para sistemas distribuídos modernos se torne uma atividade cada vez mais difícil. Dentre os fatores que impulsionam este cenário estão a evolução do *hardware* nas últimas décadas e a explosão do uso da Web. Um padrão comum no cenário de desenvolvimento é o constante aumento da complexidade dos sistemas, uma tendência que é acelerada pela adaptação de soluções em rede com ou sem fios e pela variedade de dispositivos computacionais distribuídos no ambiente físico, praticamente invisíveis aos utilizadores [4].

Em sistemas de larga escala e em *softwares* mais complexos como as aplicações distribuídas, é difícil investigar e detectar os estilos arquiteturais [16]. Essa investigação geralmente é feita manualmente e é um trabalho extremamente difícil segundo Songpon et al. [16]. No cenário atual de desenvolvimento de *software*, já existem pesquisas ressaltando a importância da detecção automatizada de estilos arquiteturais presentes no *software*, conforme apresentado na Seção V. A principal contribuição da detecção automática de estilos é melhorar a análise e compreensão da arquitetura definida para o *software*. Desvios e erosões arquiteturais são identificadas de forma mais eficaz e como consequência, garante que o produto gerado possui os atributos de qualidade induzidos pelos estilos arquiteturais presentes na arquitetura. No projeto de *software* complexos, mais tempo e esforço são gastos na verificação dos atributos de qualidade do que na construção. Muitas técnicas têm sido pesquisadas para reduzir e aliviar os esforços na verificação do *software*, aumentando a sua cobertura.

A. Representando Estilos Arquiteturais como Gramática de Grafos

Segundo Songpon et al. [16], as gramáticas de grafos fornecem uma base adequada para a reconfiguração da arquitetura e verificação de estilos arquiteturais. Elas são uma linguagem de especificação formal e precisa que ajuda a representar a arquitetura do *software* e fornece um caminho promissor para análise e evolução da arquitetura do sistema. Qualquer estilo arquitetural pode ser representado através de gramáticas de grafos [16]. Nessa representação, os elementos básicos da arquitetura como componente, conector e interface são facilmente mapeados para os elementos correspondentes num grafo onde: cada nó pode representar um componente ou conector; e cada aresta define pontos de interação entre componentes e conectores [17].

Tipicamente, uma gramática de grafo é um sistema de reescrita de um grafo, e a linguagem de grafo formal é definida para enumerar todos os grafos possíveis a partir de um grafo inicial [16]. Ela é composta por um conjunto de regras para reescrever grafos chamada de produções [16]. Uma produção (p) pode ser representada na forma: $p : L \rightarrow R$, onde L é um grafo (lado esquerdo – L) e R é o grafo (lado direito – R) derivado após a aplicação da produção. Estas produções são formadas por grafos e denotam que o grafo L está contido no grafo inicial. Uma vez identificada a relação entre o lado esquerdo da regra e o grafo inicial, a regra é aplicada gerando a transformação que gera um novo estado do sistema, onde o lado direito da regra foi aplicado ao grafo inicial, conforme ilustrado na Figura 5.

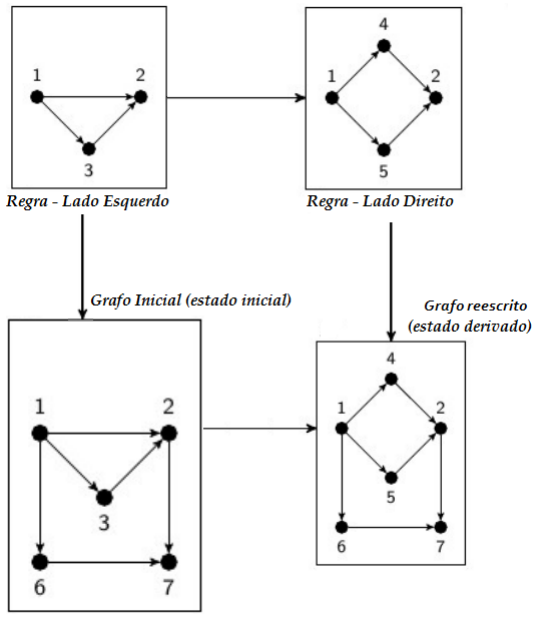

Figura 5: Visão do funcionamento das gramáticas de grafos [18].

As gramáticas de grafos existem a quase 40 anos e muitos pesquisadores definiram suas próprias gramáticas de grafos [7]. Algumas gramáticas de grafos são amplamente utilizadas em projetos e desenvolvimento de *software*, exemplo: AGG (*Attributed Graph Grammar*) [19], LGG (*Layered Graph Grammar*) [20] e RGG (*Reserved Graph Grammar*) [21]. Gramáticas de grafos são consideradas uma linguagem atraente de especificação, por que além de serem formais são baseadas em um mecanismo simples mas poderoso para descrever comportamentos. Sua representação visual permite um entendimento mais rápido e claro sobre a especificação. Por todas estes atrativos, as gramáticas de grafos têm sido bastante usadas para representação de estilos arquiteturais.

B. Verificação de Propriedades em Gramática de Grafos via Model Checking

No desenvolvimento de sistemas distribuídos um dos requisitos mais importantes durante a fase de construção é a garantia de propriedades – garantir que o sistema apresenta certas características necessárias para o seu correto funcionamento [22]. Para a verificação formal de propriedades sobre um sistema é necessário que o seu comportamento esteja descrito por um modelo matemático, ou seja, já na fase de especificação deve-se usar uma linguagem com sintaxe e semântica formais. No contexto de sistemas distribuídos, a escolha de gramáticas de grafos como linguagem formal é bastante adequada devido à complexidade desses tipos de sistemas, envolvendo vários tipos de elementos arquiteturais e diferentes tipos de relações entre eles. Uma vez desenvolvida, a especificação formal do sistema pode ser usada como base para a comprovação das propriedades da especificação.

A verificação de propriedades do sistema com gramáticas de grafos consiste na aplicação de um conjunto de regras, da gramática especificada, em um grafo inicial representando o sistema. Esse conjunto de regras especifica as mudanças de estado que podem ocorrer a partir do grafo inicial. A verificação de propriedades com gramáticas de grafos permite identificar quais regras são aplicáveis ao sistema e quais nunca serão aplicáveis. Existem técnicas automatizadas para a verificação de propriedades, a exemplo o *model checking*.

O *model checking* é uma técnica de verificação formal que consiste na representação de um sistema por meio de um modelo finito de estados o qual serão analisados para a determinação de sua conformidade com determinadas propriedades, e estas são expressas como fórmulas em lógica temporal [4]. É uma técnica de verificação que explora todos os possíveis estados do sistema usando força bruta. Semelhante a um programa de computador de xadrez – que verifica todos os movimentos possíveis – a técnica de *model checking* examina todos os cenários possíveis do sistema de uma forma sistemática. Com isso, pode ser mostrado se um dado sistema/modelo verdadeiramente satisfaz uma determinada propriedade.

As técnicas de *model checking* são muito usadas para testar automaticamente se um modelo arquitetural atende a uma dada especificação. Neste caso, tanto a arquitetura do *software* quanto os requisitos que ele deve implementar devem estar formuladas em alguma notação precisa, a exemplo gramáticas de grafos [4]. Em projetos de *softwares* complexos, como os de sistemas distribuídos, mais tempo e esforço são gastos com a verificação da arquitetura do que com a construção do *software* [4]. Muitas técnicas são pesquisadas para reduzir o tempo gasto com a verificação de modo a garantir aderência do *software* aos atributos de qualidade desejados. Os métodos formais oferecem um grande potencial para se obter uma verificação de modelo arquitetural mais eficaz e reduzir o tempo de verificação.

Um dos atrativos das técnicas de *model checking* está no fato de ser uma atividade completamente automática [4], que permite a verificação das propriedades desejadas de comportamento do sistema com base numa inspeção sistemática de todos os estados do modelo. O modelo equivale a uma representação do sistema em linguagem formal, um grafo inicial, e as propriedades, representadas por um conjunto de produções da gramática de grafos. O espaço de estados representa todas as possíveis arquiteturas em conformidade com o estilo arquitetural modelado pela gramática de grafos. Além disso, ferramentas de *model checking* tem sido amplamente utilizadas por um grande número de aplicações industriais bem sucedidas [4]. Segundo Baier et al. [4], as propriedades a serem validadas pelo *model checking* podem ser muito

elementares, por exemplo, um sistema nunca deve ser capaz de chegar a um estado em que nenhum progresso pode ser feito (cenário de *deadlock*). Contudo, estas propriedades precisam estar descritas em um conjunto de regras de transformações e constituem a base da atividade de *model checking*. Um defeito é encontrado toda vez que o sistema modelo não cumpre uma das propriedades especificadas, ou seja, se a propriedade não pode ser aplicada ao grafo. O sistema é considerado correto sempre que satisfaz todas as propriedades descritas. A Figura 6 apresenta uma visão simplificada do *model checking*, detalhado logo abaixo:

- *System Modeling*: representa a arquitetura do sistema (modelo) que passou por um processo de modelagem usando linguagem formal;
- *Property Specification*: representa a propriedade que será checada. Corresponde ao requisito descrito em linguagem formal, que o sistema deve atender;
- *Model Checking*: implementa a verificação de propriedades que pode ter resultado *satisfied* – sistema satisfaz propriedade – ou *violated+counterexample* quando indica falha devido a violação da propriedade no sistema;
- *Simulation*: permite a geração de um espaço com universo de estados possíveis do sistema, a partir da aplicação das regras que representam as propriedades checadas. Os estados que violam algumas destas regras são identificados.

Segundo Baier et al. [4], a utilização de *model checking* permite examinar todas as informações relevantes sobre os estados do sistema para verificar se satisfazem às propriedades desejadas. Se for encontrado um estado que viole a especificação formal de alguma propriedade, o simulador do *model checking* irá fornecer um contra-exemplo que indica como o modelo pode atingir o estado indesejado. Neste contra-exemplo é descrito o caminho de execução que leva a partir do estado inicial a um estado que viola a propriedade que está sendo verificada.

IV. Mecanismo de Detecção Proposto

Motivado pela necessidade de novos métodos que atestem a qualidade do *software*, este trabalho propõe um mecanismo para verificação do *software* através da detecção dos estilos arquiteturais presentes na arquitetura do sistema. O mecanismo de detecção consiste em verificar nos sistemas distribuídos, os estilos arquiteturais implementados e como isso induzir no sistema as propriedades relacionados ao estilo arquitetural detectado. Para tal finalidade foram usadas técnicas de *model checking*, gramáticas de grafos e desenvolvimento de um mecanismo que utiliza a combinação destas técnicas, para implementar a detecção de estilos arquiteturais em sistemas distribuídos.

Os estilos arquiteturais a serem verificados devem estar descritos em gramáticas de grafos. Essa gramática deve ser aplicada ao grafo inicial do sistema modelo – com os elementos arquiteturais básicos da arquitetura – e através de técnicas de *model checking* é feita a simulação de todos os possíveis estados resultantes da aplicação das produções da gramática sobre o sistema modelo. Com a utilização de gramáticas de grafos para representação dos estilos arquiteturais é possível verificar propriedades especificas presentes no grafo do sistema como: a alcançabilidade, que indica quais tipos de vértices estão presentes; a aplicabilidade, que identifica quais regras são aplicáveis no sistema e quais nunca serão aplicadas; e o conflito, que encontra situações em que concorrentemente duas regras podem ser aplicadas. Segundo Rensink et al. [23], a verificação de sistemas complexos como aplicações distribuídas é um vasto campo a ser explorado. Muitos pesquisadores estão estudando a aplicação de *model checking* para a verificação de arquitetura do *software*. Já existem algumas ferramentas que implementam *model checking* a exemplo o GROOVE (*Groove GRaphs for Object-Oriented Verification*) [4]. A Figura 7 apresenta a tela inicial do GROOVE.

O GROOVE é um conjunto de componentes destinados à simulação e análise de transformações em grafos bastante versátil e simples de usar [23]. Ele é uma ferramenta acadêmica *open source* mantida por um grupo de pesquisa do departamento de ciência da computação da universidade de *Twente* na Holanda há mais de 10 anos, e vem sendo melhorada com novas extensões ou modificações adicionadas constantemente. O GROOVE permite simular a aplicação das produções de uma gramática em um grafo inicial e gerar um espaço de estados com todos os grafos resultantes e derivações possíveis. Para a implementação do mecanismos de detecção apresentado por este trabalho, foi adotada a utilização do GROOVE para as seguintes tarefas:

- Criação das gramáticas de grafos que representam cada estilo arquitetural, com suas regras, elementos e restrições arquiteturais. Nesta tarefa são considerados os seguintes estilos arquiteturais: *Client-Server*, *Peer-To-Peer*, REST e *MapReduce*;
- Criação do grafo inicial que representa o sistema modelo com os elementos arquiteturais e restrições correspondentes a cada estilo arquitetural;
- Criação do grafo que representa a arquitetura final do sistema a ser verificado;
- Geração de um espaço com universo de estados possíveis do sistema, a partir da aplicação das regras de transformação de grafos. Cada estado é um grafo que representa uma nova estrutura do sistema. Os estados que violam algumas destas regras são identificados.

Entretanto, para alcançar a meta do mecanismo de detecção, que consiste em detectar estilos arquiteturais em uma dada arquitetura, foi necessário implementar uma nova funcionalidade no GROOVE. O objetivo deste novo recurso é verificar se uma dada arquitetura final de entrada está presente na árvore de derivação da gramática de grafos, gerada pelo simulador do GROOVE. Se o grafo de entrada for encontrado, podemos afirmar com certeza que a arquitetura está em conformidade com o estilo. Caso contrário, nada podemos afirmar, pois pode ser que o simulador do GROOVE não tenha sido executado tempo o bastante para gerar um maior numero de estados resultantes que equivale à árvore de derivação da gramática.

Todos os componentes do GROOVE são escritos na linguagem Java e, portanto, ele pode ser executado em qualquer plataforma com uma máquina virtual Java 6. Por se tratar de uma ferramenta *open source*, foi possível obter o código

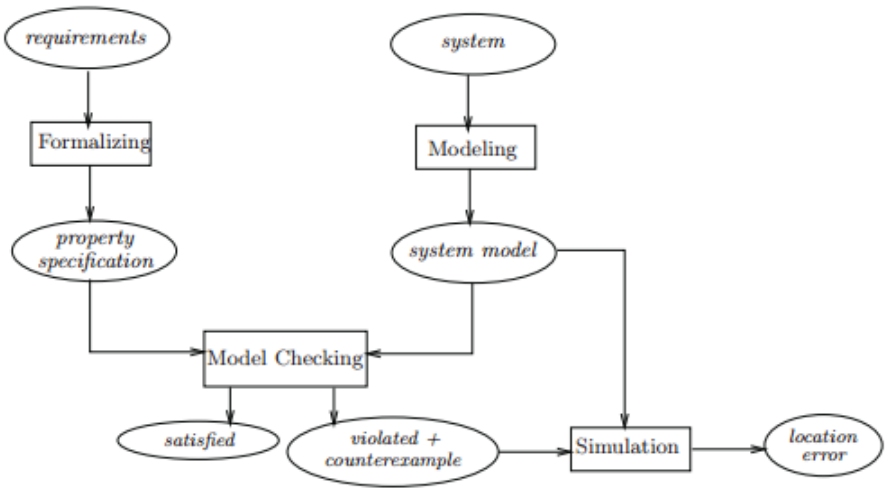

Figura 6: Visão simplificada do *model checking* [4].

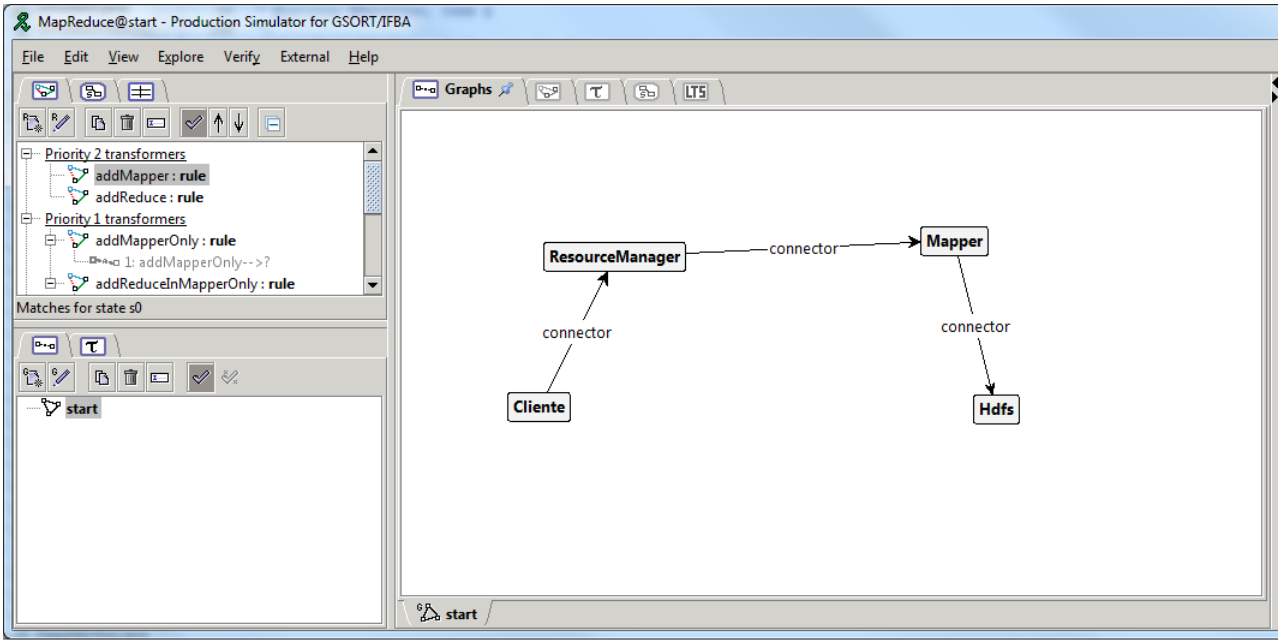

Figura 7: Tela inicial do GROOVE implementado com base na versão 5.5.0.

fonte do GROOVE para implementação da funcionalidade de detecção *check graph on full state space*. Para implementação da nova funcionalidade foi necessário entender a arquitetura do GROOVE que é composta pelos seguintes componentes, onde cada um é compactado em um arquivo com extensão jar (*Java ARchive*):

- *Simulator* (.jar) - Componente baseado em GUI (*Graphical User Interface*) do Java, que permite construir ou editar visualmente regras de transformações e grafos. Além disso, o *simulator* permite ao utilizador explorar de forma interativa a aplicação manual de regras sobre um grafo;
- *Generator* (.jar) - Componente de linha de comando que permite simular aplicações de regras de transformações em grafos sem a necessidade da interface GUI;
- *Viewer* (.jar) - *Interface* GUI que representa o editor do GROOVE para visualizar, editar e criar grafos e regras de transformação sobre os grafos. Usado pelo componente *Simulator*.
- *ModelChecker* (.jar) - Componente para verificação de modelo que permite verificar o espaço de estados gerados, a partir do conjunto de produções aplicadas ao grafo inicial. Esse componente permite executar uma busca exaustiva em cada estado gerado;
- *PrologChecker* (.jar) - Verificador de modelos que suporta fórmulas em lógica temporal, como LTL (*Linear*

Temporal Logic) ou CTL (*Computation Tree Logic*), permite aplicar uma fórmula CTL ou LTL em todos os estados gerados, ou em um estado específico;

- *Imager* (.jar) - Componente para a conversão de grafos em imagens.

Para implementação da funcionalidade *check graph on full state space* no GROOVE, foi utilizado o eclise como IDE de programação e Java 1.6. O projeto do GROOVE é muito bem documentado e conta com uma equipe de desenvolvimento bastante ativa. No início do estudo para elaboração deste trabalho o GROOVE estava na versão 5.3.3, atualmente o GROOVE está na 5.5.2 sendo que existe uma versão em teste para tratar os recentes erros apontados pelos membros do grupo de pesquisa. A implementação do mecanismo de detecção no GROOVE foi realizada sobre a versão 5.5.0, pois essa era a versão atual quando foi iniciada a construção das gramáticas de grafos para os estilos arquiteturais a serem detectados. Para evitar possíveis problemas de compatibilidade das gramáticas de grafos já criadas na versão 5.5.0, a implementação do mecanismo de detecção foi realizada sobre essa versão. Para entender a melhoria implementada no GROOVE, foi elaborado um diagrama das classes relacionadas com a implementação da funcionalidade *check graph on full state space*. Os itens destacados de laranja foram modificados, o item de verde foi criado e o item destacado de azul representa a principal classe do componente *Simulator* que também foi modificada, conforme ilustrado na Figura 8.

A funcionalidade de *check graph on full state space* é disponibilizada após ser gerado no *Simulator* o espaço com os estados resultantes da aplicação das gramáticas sobre o grafo inicial. Ao acessar o menu *Verify*, opção *check graph on full state space* é aberta uma janela para indicar o caminho do grafo que representa o sistema a ser analisado. Ao selecionar o grafo é iniciada a busca no espaço de estados, onde cada estado representa um grafo que será comparado ao grafo selecionado no acesso à funcionalidade *check graph on full state space*. A lógica de verificação está implementada no método *exploreForGraph* e consiste em percorrer a coleção de espaços resultante testando se o grafo que representa o estado da interação atual é isomórfico ao grafo de entrada. Para isso é usado o método *areIsomorphic* da *classe IsoChecker*, conforme pode ser visualizado na Figura 9. Quando é encontrado isomorfismo a interação é interrompida e um alerta é exibido, indicando o identificado do estado onde foi encontrada equivalência.

No mecanismo para detecção de estilos arquiteturais voltados para sistemas distribuídos, cada aplicação de uma regra de produção ou transformação gera um novo estado para o grafo inicial. Na regra estão as condições que devem ser satisfeitas para que a regra possa ser aplicada ao grafo. Essas condições são formuladas em lógica temporal, tipo de lógica onde é possível representar e raciocinar sobre uma sequência de estados alcançáveis do sistema [24]. A principal diferença da lógica temporal para a convencional, está no fato da lógica temporal poder apresentar valores verdade que podem variar com o tempo, ou seja, a fórmula pode ser verdadeira em alguns estados e falsa em outros [24]. Na lógica temporal, pode-se usar operadores modais que permitem expressar se uma propriedade é válida em todo o modelo ou em parte dele. Os tipos de lógica temporal são classificados de acordo com a estrutura do modelo de tempo assumido, podendo ser linear ou ramificada. Tradicionalmente existem duas formalizações de lógica temporal no contexto de verificação de modelos: *Lógica Temporal Linear*, onde os operadores descrevem propriedades presentes em todos os estados possíveis do sistema e *Computation Tree Logic*, onde os operadores temporais inferem sobre diferentes trajetórias a partir de um estado inicial. Segundo [4], na abordagem usando fórmulas LTL é considerado que uma propriedade deve ser quantificada para todas as execuções do sistema. Enquanto na abordagem em CTL, por sua vez, considera que uma propriedade pode ser quantificada para uma ou todas as execuções do sistema.

Para funcionamento do mecanismo de detecção são necessárias cinco etapas:

1) Modelagem do grafo inicial correspondente à gramática - Consiste na representação formal modelo (grafo inicial) para que a partir deste modelo seja possível obter todos os comportamentos possíveis do sistema. Para cada estilo arquitetural deve existir um modelo que será usado para validar a aplicação da gramática de grafos que representa o estilo arquitetural verificado e gerar os estados resultantes.
2) Modelagem do sistema a ser detectado - Consiste na representação formal do *software* (grafo de entrada) que representa a arquitetura final do sistema.
3) Especificação - Esta etapa consiste em especificar as propriedades desejáveis do sistema que implementa o estilo arquitetural a ser validado. Estas propriedades são descritas formalmente com regras de produções ou transformações para gramáticas de grafos. Para a implementação das regras que compõem a gramática de grafos implementada são utilizadas: fórmulas LTL (*Lógica Temporal Linear*) para construir as validações de restrições em cada estilo arquitetural a ser detectado; e fórmulas CTL (*Computation Tree Logic*) para gerar as variações válidas para arquitetura do sistema representado pelo grafo inicial. Cada estilo arquitetural implementa determinado conjunto de regras e tem sua gramática de grafos correspondente implementada no GROOVE.
4) Verificação: Nessa etapa o modelo e as propriedades especificadas são submetidas aos componentes *ModelChecker* e *simulator* do GROOVE para verificar a aplicabilidade das produções e gerar o espaço com os estados resultantes a partir da aplicação das produções sobre o grafo inicial que representa o modelo do sistema.
5) Detecção do sistema: A funcionalidade de *check graph on full state space* é acionada para ler o grafo que representa o sistema a ser verificado e realizar uma busca exaustiva no espaço de estados resultantes, conforme ilustrado na Figura 10.

A principal contribuição deste trabalho é o mecanismo para detecção de estilos arquiteturais em sistemas distribuídos. Para construção do mecanismo, além da implementação de um funcionalidade no GROOVE, foi necessário a criação das gramáticas com suas regras de transformações e validações das restrições. De maneira geral, as propriedades correspondentes ao estilo arquitetural a ser detectado foram representadas por regras de produção da gramática de grafos correspondente.

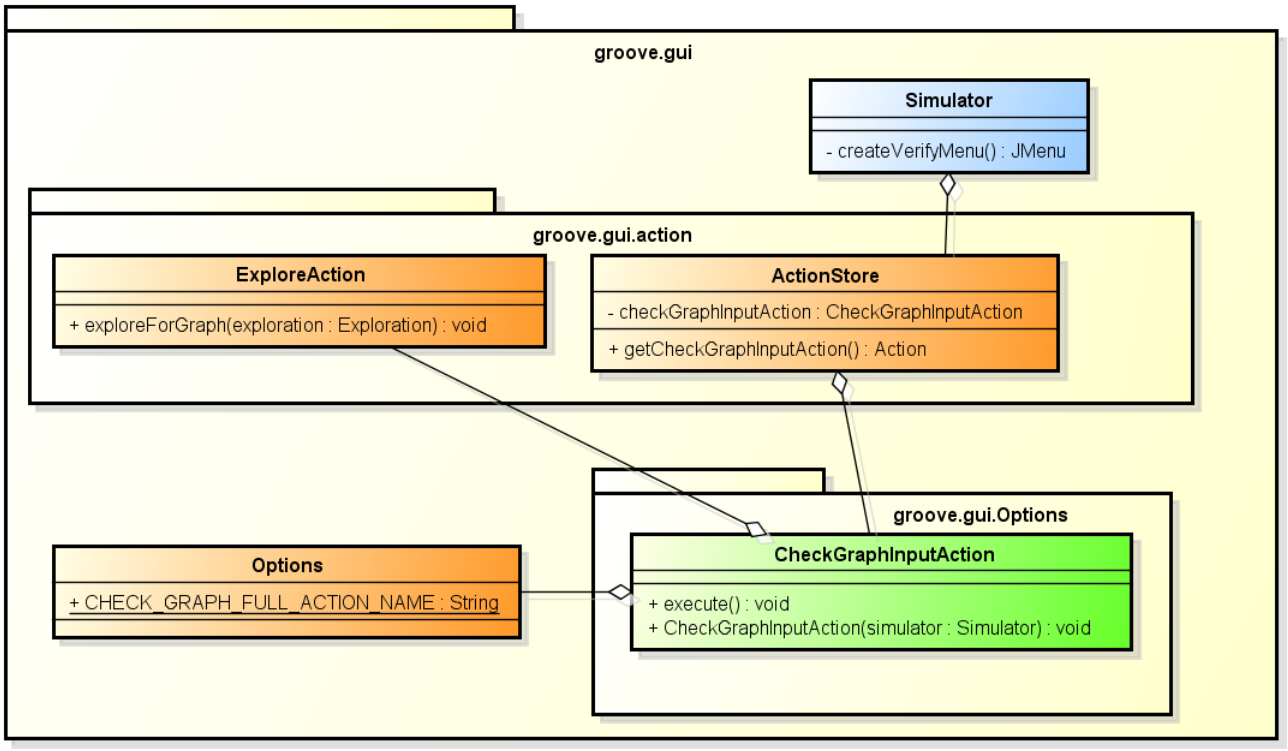

Figura 8: Diagrama de classes da funcionalidade *check graph on full state space*.

```
public void exploreForGraph(Exploration exploration) {
    if (getSimulatorModel().getExploration() != null
            && getSimulatorModel().getExploration().getGTS() != null) {

        // Read Graph for Check
        HostGraph hostGraphInput = getHostGraphInput();

        Collection<GraphState> coll = getSimulatorModel().getExploration()
                .getGTS().getStates();
        IsoChecker isoChecker = IsoChecker.getInstance(true);
        boolean areIsomorphic = false;

        // Checks the general states
        for (GraphState graphState : coll) {
            HostGraph hostGraph = graphState.getGraph();
            areIsomorphic = isoChecker.areIsomorphic(hostGraphInput,
                    hostGraph);
            if (areIsomorphic) {

                String message = String
                        .format("Isomorphism found in state '%s' in the set of states",
                                hostGraph.getName());
                JOptionPane.showMessageDialog(getFrame(), message);
                break;
            }
        }
        if (!areIsomorphic) {
            String message = String
                    .format("Isomorphism not found in the set of states");
            JOptionPane.showMessageDialog(getFrame(), message);
        }
    }
}
```

Figura 9: Lógica implementada pelo *check graph on full state space*.

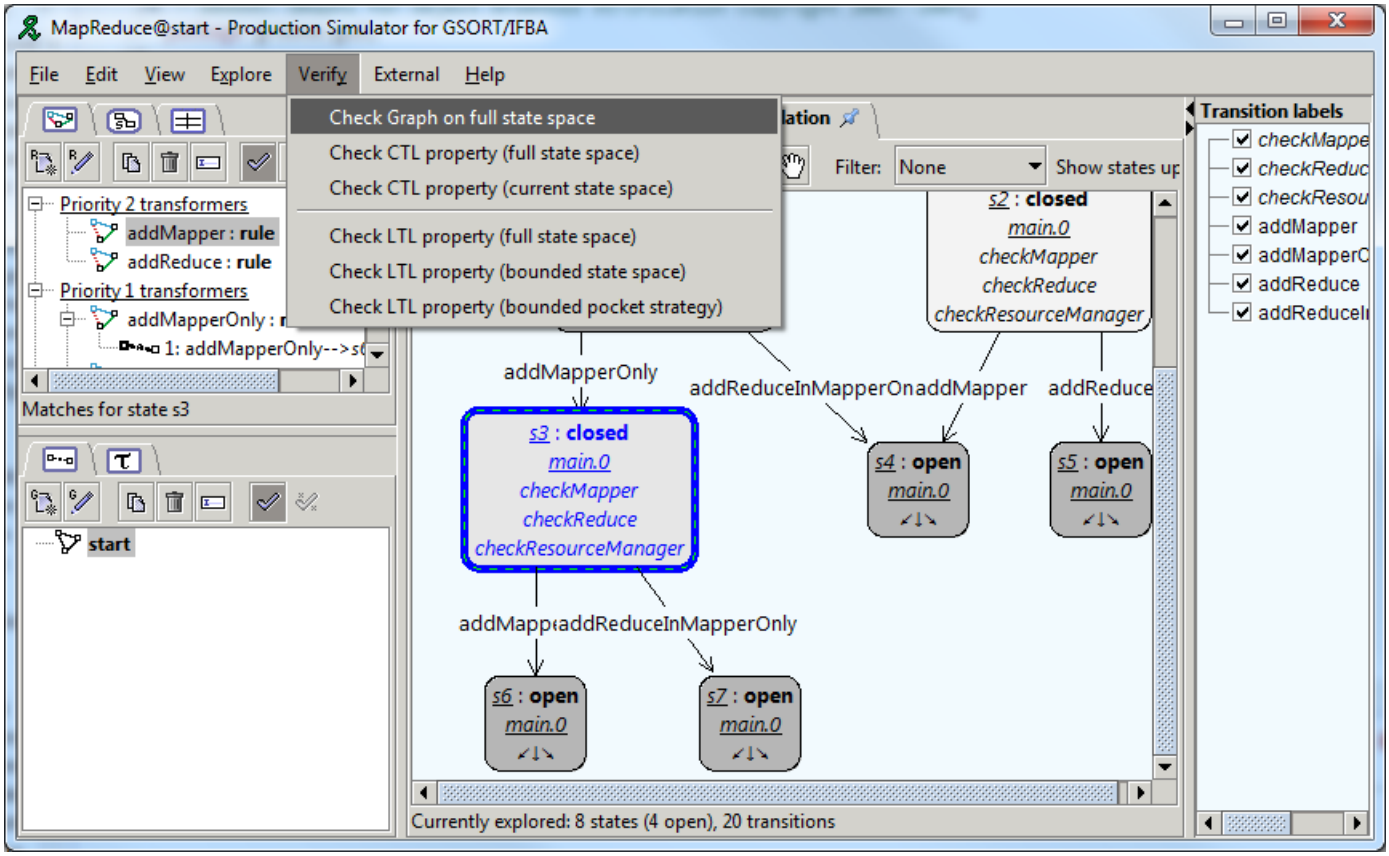

Figura 10: Tela do GROOVE - Funcionalidade *check graph on full state space*.

Cada regra foi implementada em lógica temporal e define os requisitos e restrições para a conformidade com o estilo arquitetural a ser detectado.

O mecanismo de detecção implementado envolve estilos arquiteturais voltados para sistemas distribuídos. Dentro deste contexto foram selecionados os principais estilos arquiteturais usados nesses tipos de aplicações: *Client-Server*, *Peer-To-Peer*, REST e *MapReduce*. O primeiro passo para implementação da gramática de grafos consistiu na especificação dos tipos de elementos e relações presentes na organização arquitetural do estilo arquitetural. O conjunto das restrições sobre como estes elementos podem ser usados pelo sistema está sendo representado por produções da gramática de grafos. Para cada estilo foi definido um modelo de sistema, representado como grafo inicial, que implementa a arquitetura modelo correspondente ao estilo arquitetural. Nas próximas subseções é feito um detalhamento de cada gramática criada para representar os estilos arquiteturais que compõe o escopo da detecção.

A. Gramática de Grafos para Estilo Arquitetural MapReduce

Para detecção do estilo arquitetural *MapReduce* foi definido um grafo inicial que representa um sistema com arquitetura *map-only*. Neste tipo de arquitetura estão presentes apenas componentes do tipo *Mapper* [12], conforme apresentado na Figura 11. A gramática de grafos que representa este estilo considera que tanto podem ser adicionadas novos componentes *Mappers* à arquitetura, como também é válido a adição de componentes do tipo *Reduces*, desde de que nesse cenário todos os *Mappers* existentes na arquitetura passem a interagir com os componentes do tipo *Reduces*.

As produções implementadas para representação do estilo arquitetural *MapReduce* verificam a existência de componentes válidos neste tipo de arquitetura e suas respectivas funções, tipos de conectores, restrições e derivações, conforme segue:

- *addMapperOnly* - Produção criada para adicionar novos componentes do tipo *Mappers* em uma arquitetura *map-only*. Para tal finalidade verifica antes se já existe pelo menos um componente do tipo *Mapper* e nenhum componente do tipo *Reduce* para validar o cenário de *map-only* e aplicabilidade da produção, conforme ilustrado na Figura 12.
- *addReduceInMapperOnly* - Produção criada para adicionar componente do tipo *Reduce* em uma arquitetura do tipo *map-only*. Na aplicação da produção, todos os componentes do tipo *Mapper*, existentes no grafo, devem passar a interagir com o componente *Reduce* adicionado pela produção, conforme ilustrado na Figura 13. Dessa forma, a arquitetura do sistema deixa de ser do tipo *map-only*, mas continua atendendo às especificações do estilo arquitetural *MapReduce*. Além de adicionar o componente do tipo *Reduce* e vincular todos componentes do tipo *Mapper* presentes na arquitetura ao novo componente *Reduce* adicionado, a produção deve também remover as conexões antigas

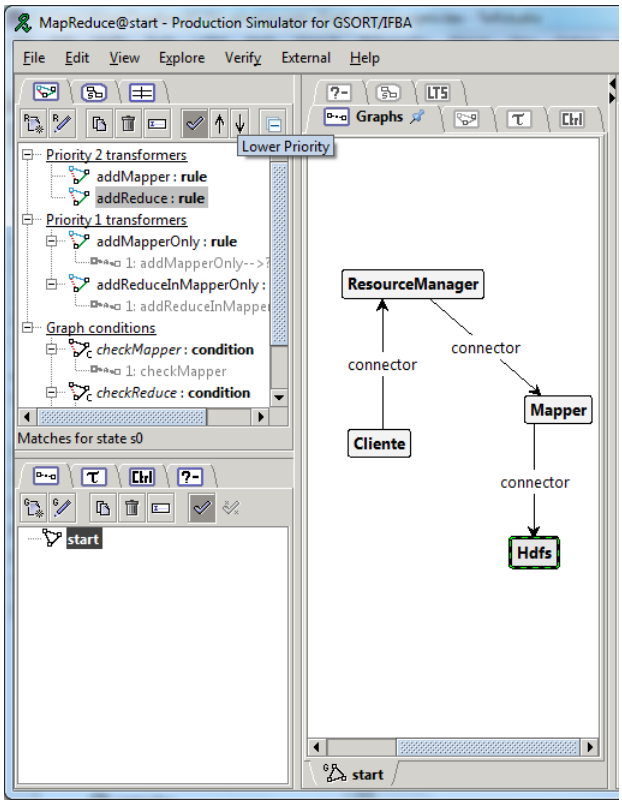

Figura 11: Grafo Inicial.

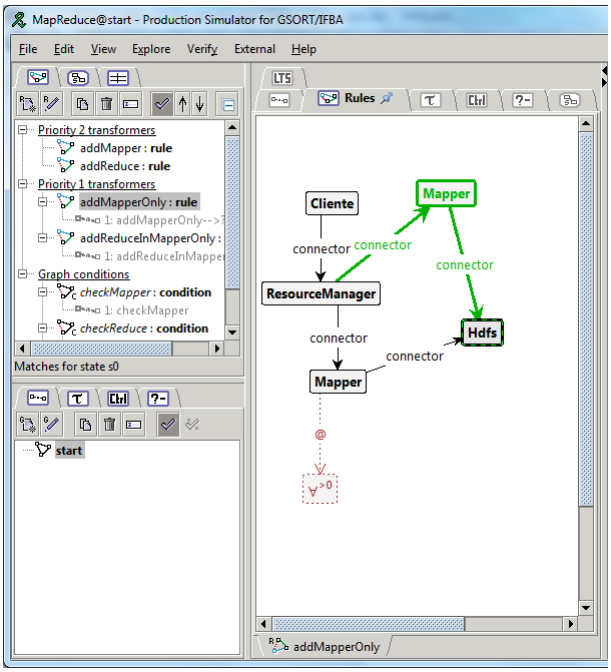

Figura 12: *addMapperOnly* - O item destacado de verde representa o novo elemento a ser inserido na arquitetura.

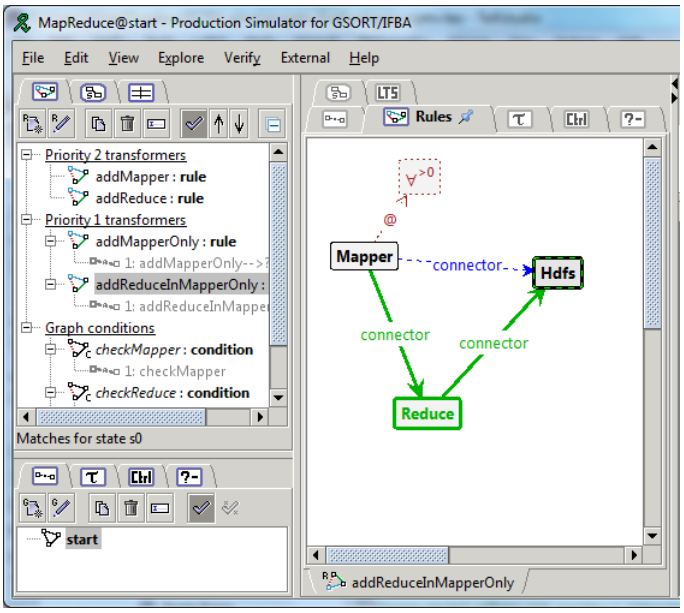

Figura 13: *addReduceInMapperOnly* - O item destacado de verde representa o novo elemento a ser inserido na arquitetura e o item destacado em azul, representa o elemento removido.

entre os *mappers* existentes e o componente HDFS.

- *addMapper* - Produção criada para adicionar um componente do tipo *Mapper* numa arquitetura onde já existem componentes do tipo *Reduce*, conforme ilustrado na Figura 14. Na aplicação desta regra, o componente *Mapper* adicionado deve interagir com todos os componentes do tipo *Reduce* existentes no grafo;
- *addReduce* - Produção criada para adicionar um componente *Reduce* numa arquitetura que não é do tipo *map-only*, conforme ilustrado na Figura 15. Na aplicação desta regra, todos os componentes do tipo *Mapper* já existentes na arquitetura, devem interagir também com o componente do tipo *Reduce* adicionado.

Além da implementação das produções para a gramática que representa o estilo arquitetural *MapReduce*, foi necessária a implementação de restrições denominadas no GROOVE como *graph conditions*. Elas validam a comunicação entre os elementos da arquitetura, tais como: um componente do tipo *Mapper* não pode-se comunicar com outro componente *Mapper* e caso exista na arquitetura um ou mais componente *Reduce*, todos os demais componentes *Mappers* devem se comunicar com os *Reduces*. Na Figura 16 é apresentado o conjunto de estados derivados do grafo inicial, representado pela Figura 11, após aplicação das produções via simulação no GROOVE. Nessa simulação é possível perceber que para o grafo inicial, dentre as quatro produções criadas apenas duas podem ser aplicadas inicialmente: *addMapperOnly* ou *addReduceInMapperOnly*. Isso significa que é válido adicionar componentes *mapper* numa arquitetura onde não existem componentes *reduces*, assim como também é válido adicionar um componente *reduce* desde que os componentes *mappers*

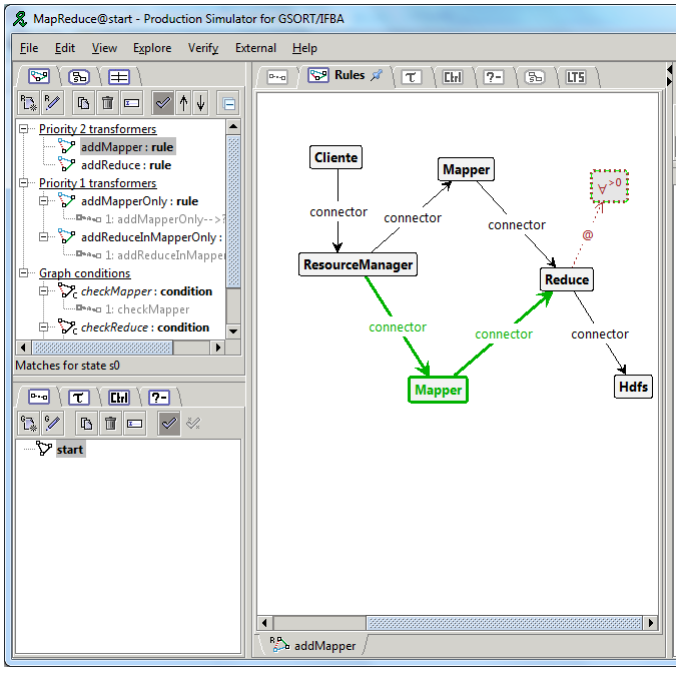

Figura 14: *addMapper* - O item destacado de verde representa o novo elemento a ser inserido na arquitetura.

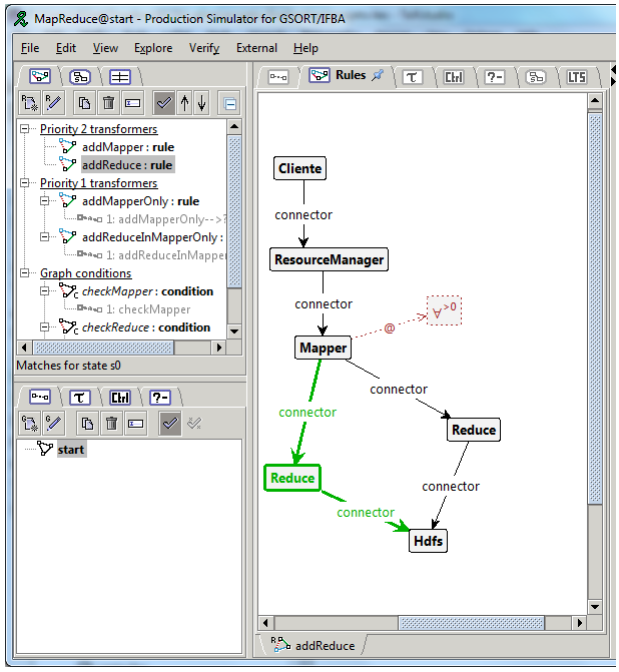

Figura 15: *addReduce* - O item destacado de verde representa o novo elemento a ser inserido na arquitetura.

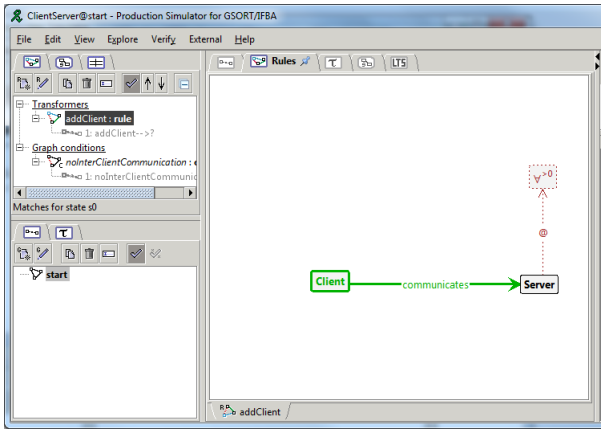

Figura 17: Regra para adicionar novo componente *Client*.

existentes passem a se comunicar como o componente *reduce* adicionado. A partir do momento que uma regra ou outra é aplicada, novas regras passam a ficar disponíveis para serem aplicadas e novas derivações são geradas. Cada estado gerado na simulação representa um grafo e possui identificadores exemplo: s1, s2, s3, s4, s5 e s6. A funcionalidade *check graph on full state space*, executa uma verificação em cada estado para validar se existe isomorfismo entre o grafo que representa o estado e o grafo que representa a arquitetura final do sistema verificado.

B. *Gramática de Grafos para Estilo Arquitetural Client-Server*

Nela foram criadas regras para validar a comunicação entre os componentes de modo a garantir que a comunicação sempre será iniciada pelo componente do tipo *Client* e nunca um *Client* irá solicitar serviços a um outro componente do tipo *Client*. Nesta gramática, a presença dos tipos de conectores e componentes do estilo arquitetural correspondente também são validados através de produções que criam novos componentes do tipo *Client* se comunicando com o componentes do tipo *Server*, conforme apresentado na Figura 17. Para validação de restrições foram implementadas *graph conditions*. Um exemplo de restrição arquitetural verificado com *Graph conditions* nessa gramática é: não deve haver comunicação entre componentes do tipo *Client*, conforme apresentado na Figura 18. Todos os estados gerados que atendem à restrição são marcados, desse modo é possível validar se existe algum estado que represente um desvio arquitetural.

C. *Gramática de Grafos para Estilo Arquitetural Peer-To-Peer*

As regras implementadas validam a presença de componentes presentes nesse tipo de arquitetura e seus conectores. Na gramática construída é considerado que pode existir na arquitetura componentes do tipo *supernode*, que são um tipo especial de *peer* com capacidade de roteamento para permite que *peers* conectados a ele, consigam conhecer um maior número de outros *peers*, conforme apresentado na Figura 19.

A gramática de grafos construída para este estilo arquitetural é mais simples quando comparada a outras gramáticas construídas como a do *MapReduce*. Isso se deve ao fato de existir menos tipos de componentes nessa arquitetura. A Figura

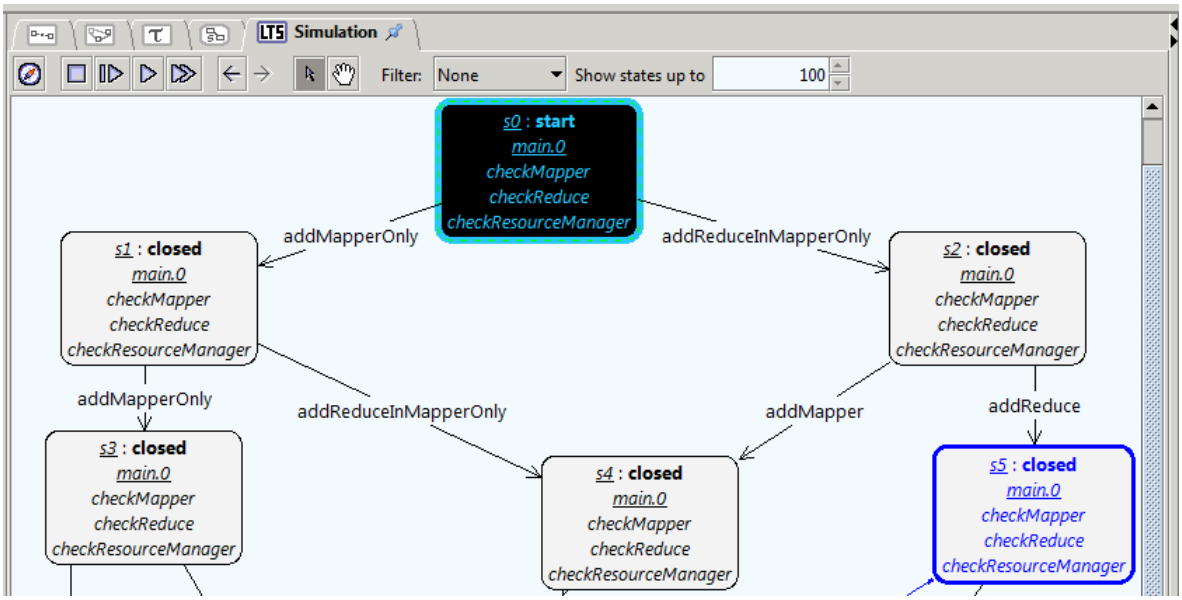

Figura 16: Visão do resultado da simulação no GROOVE - Exibição de todos os estados possíveis e as restrições quando encontradas.

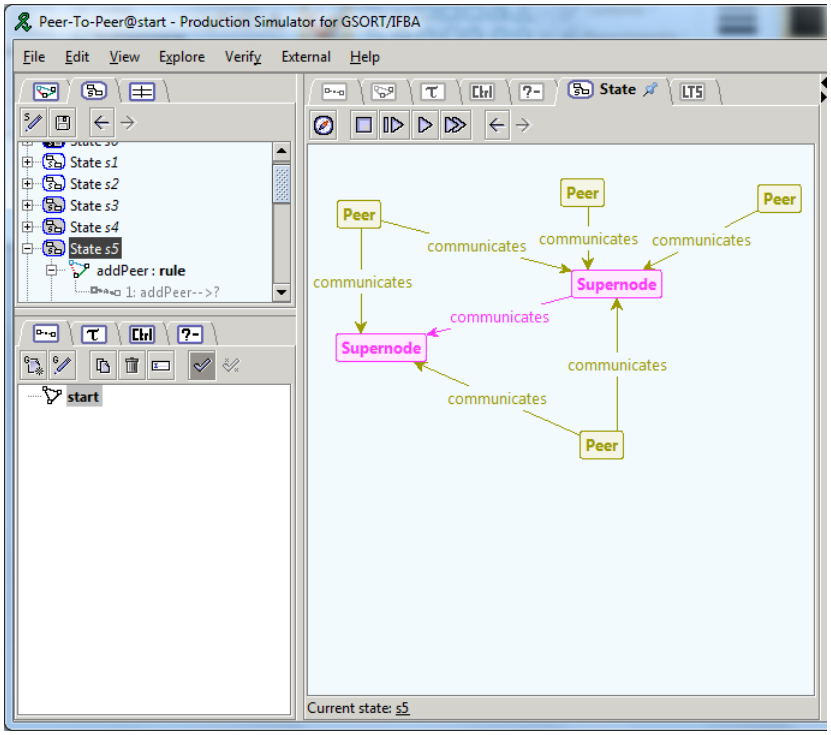

Figura 19: Simulação executada para a gramática construída para o *Peer-To-Peer*.

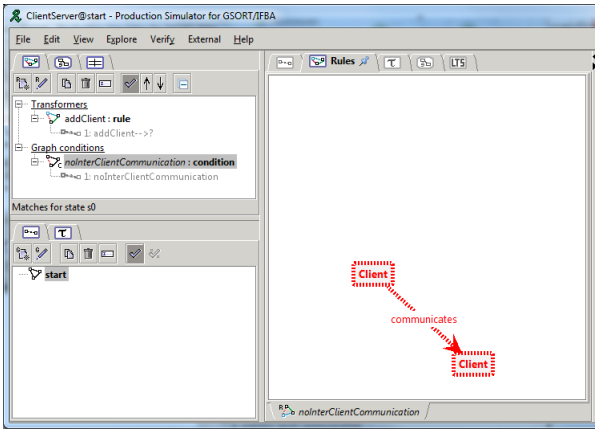

Figura 18: Produção que verifica comunicação entre componentes do tipo *Client*.

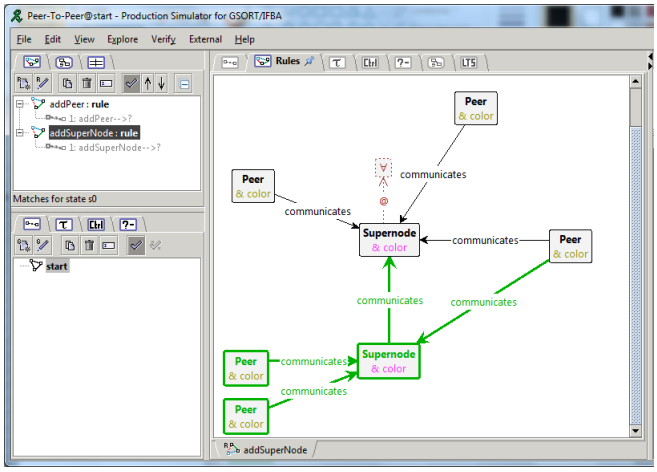

Figura 20: Produção para adicionar componentes *supernodes* à arquitetura.

20 apresenta a produção criada para adicionar componentes *supernodes* à arquitetura *Peer-To-Peer*.

D. Gramática de Grafos para Estilo Arquitetural REST

Assim como todos os estilos arquiteturais voltados para Web, o REST deve ser capaz de suportar o constante crescimento de usuários e servidores. Com essa finalidade a gramática construída possui regras para adicionar servidores e clientes. Essa gramática é muito parecida com a gramática criada para o estilo *Client-Server*, uma vez que o REST herda conceitos de outros estilos arquiteturais, principalmente do *Client-Server*. Contudo novas restrições são aplicadas aos componentes, conectores e elementos de dados para a Web. Essas restrições são implementas em cada uma das produções criadas para está gramática, conforme segue:

- *addClient*: nessa produção, criada para adicionar um novo cliente à arquitetura REST, será incrementado um contador cada vez que um novo cliente se conecta ao servidor. Esse controle ocorre para realizar um balanceamento de carga de modo a evitar que o servidor fique sobrecarregado, ou seja, se o servidor já tiver

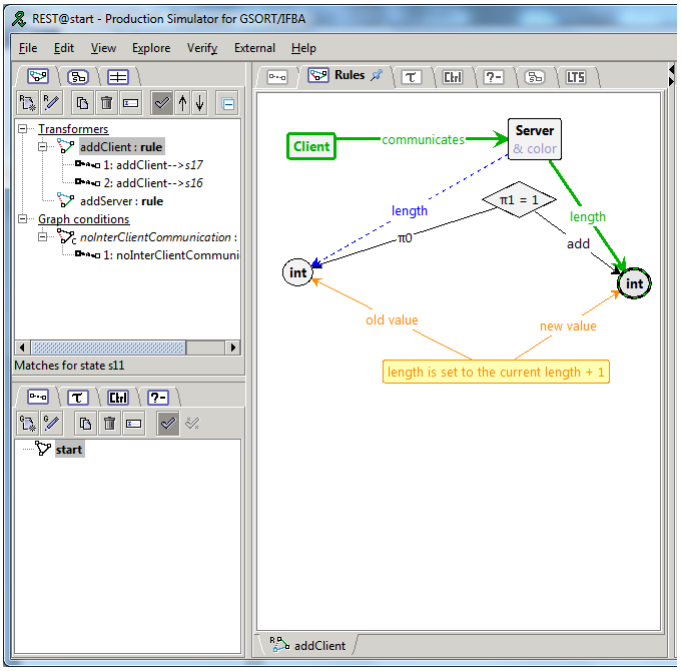

Figura 21: Produção para adicionar clientes e incrementar contador do servidor.

um número elevado de clientes conectados, essa regra não será aplicada e uma nova regra será habilitada para lidar com esse cenário, conforme apresentado na Figura 21;

As gramáticas de grafos criadas, combinadas com as técnicas de verificação de modelos (*model checking*) mais a funcionalidade *check graph on full state space* implementada no GROOVE, compõem o mecanismo de detecção apresentado neste trabalho. Seu papel consiste em ser um apoio à atividade de análise arquitetural. Nesse trabalho cada parte que compõe o mecanismo de detecção de estilos arquiteturais em sistemas distribuídos foi detalhada isoladamente para melhor explicar sua função, como foi feita sua concepção e implementação. Para finalizar o entendimento sobre o funcionamento do mecanismo em sua totalidade, a Figura 22, representa a visão geral do mecanismo. A funcionalidade *check graph on full state space* percorre o espaço de estados gerados pelo simulador e verifica se em algum dos estados existe isomorfismo entre o grafo que representa o estado e o grafo que representa o sistema verificado. Quando encontrado isomorfismo é retornado para o usuário um alerta com o identificador do estado onde a equivalência foi encontrada. Quando não encontrado isomorfismo em nenhum dos estados, nada pode ser declarado uma vez que o tempo de execução do simulador para gerar o espaço de estados resultantes interfere na quantidade de estados gerados. Isso quer dizer que o tempo de execução do simulador pode não ter sido o suficiente para gerar todos os estados do espaço. O único pré-requisito do mecanismo é que: *i*) o sistema verificado, assim como os estilos arquiteturais, estejam representados por linguagem de grafo formal; *ii*) o espaço de espaços resultantes tenha sido gerado no GROOVE.

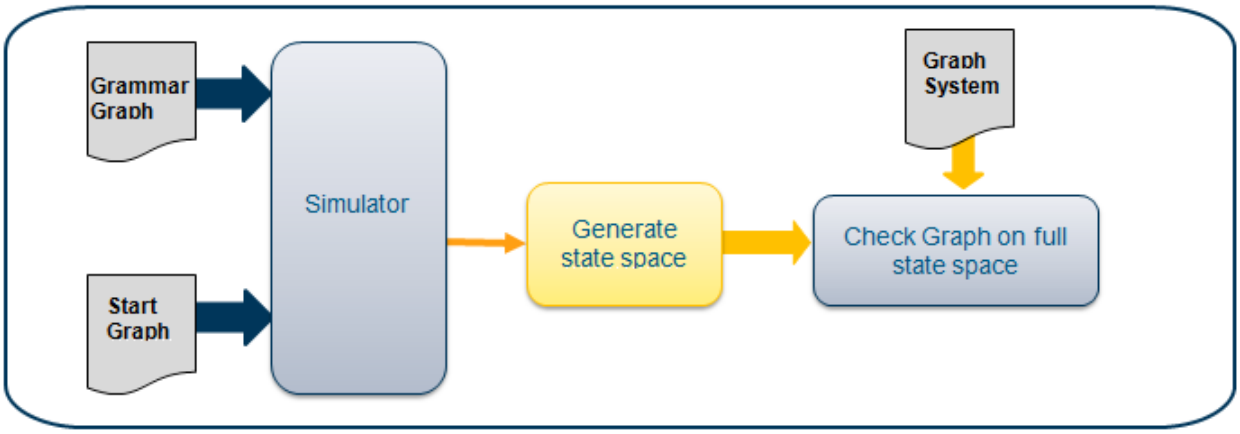

Figura 22: Visão geral do mecanismo de detecção.

V. Avaliação

A avaliação do mecanismo de detecção de estilos arquiteturais consistiu em etapas a serem executadas em ordem para garantir uma avaliação eficaz. Em todas as etapas o próprio GROOVE viabilizou a avaliação gerando resultados para análise e comparações. As etapas definidas para avaliação são detalhadas nas subseções seguintes.

A. Corretude das Gramáticas de Grafos

Esta foi a primeira etapa executada. Ela consistiu em validar a corretude das gramáticas criadas. Para isso, foi criado para cada gramática um grafo inicial e regras para produção de transformações com os elementos e restrições arquiteturais que representam o estilo arquitetural. Com o GROOVE foi possível gerar vários espaços de estados e verificar se em cada um desses estados houve violação de alguma restrição. No GROOVE quando um estado viola uma restrição o mesmo não é rotulado com a *graph conditions* que valida a restrição.

B. Verificação da arquitetura

Para avaliação dessa etapa, foram feitos testes para cada estilo arquitetural a ser detectado. Em cada caso foi submetido um grafo que representa o sistema que implementava o estilo e outro grafo que não implementava. Em todas as execuções do mecanismo de detecção o resultado foi correto quando existia de fato o estilo implementado. Contudo, nos casos em que foi submetida para detecção um grafo de sistema onde o estilo propositalmente não estava implementado, a ferramente retorna que não foi encontrado isomorfismo em nenhum dos estados verificados. É importante ressaltar que o tempo de simulação interfere diretamente na detecção, pois quanto menor o tempo da simulação, menor o espaço de estados resultante. O tempo de simulação no GROOVE pode ser limitado à quantidade de espaços gerados, exemplo: executar simulação até alcançar o máximo de 5000 espaços gerados.

VI. Trabalhos correlatos

A análise arquitetural tem sido tema em muitos trabalhos de pesquisa [2]. Nas subseções seguintes, alguns destes trabalhos serão brevemente detalhados, destacando suas motivações, técnicas utilizadas e pontos relevantes. No final será feito um comparativo sobre estes trabalhos correlatos e o trabalho apresentado neste artigo.

A. Detecting Architecture Erosion by Design Decision of Architectural Pattern [25]

O trabalho de Zhang [25] apresenta uma abordagem para detecção de erosão arquitetural em projeto de *software*. Ele propõe uma ferramenta de apoio à análise arquitetural para avaliar se o código implementado no projeto está de acordo com o que é esperado para o padrão arquitetural definido.

1) Motivação: A principal motivação para o trabalho de Zhang [25] é o argumento de que a melhor arquitetura é inútil se o código implementado não segue a mesma. No entanto, é difícil garantir que cada arquitetura projetada será fielmente transformada em código fonte.

2) Técnica usada: Para auxiliar o processo de análise arquitetural, Zhang [25] definiu um meta-modelo de padrão de projeto com um conjunto mínimo de elementos construtivos para caracterizar o padrão de projeto. Com base neste meta-modelo é gerado código em OCL (*Object Constraint Language*) onde as restrições referentes ao padrão arquitetural são expressas.

3) Ponto Fraco: Um padrão arquitetural deve ser o mais genérico possível para resolver problemas de contextos específicos [6]. Devido a essa necessidade, os padrões não definem completamente a arquitetura do sistema. Como o trabalho de Zhang [25] trata de detecção de padrões arquiteturais, ele não é tão efetivo no requisito compreensão e análise arquitetural. Além disso, o mecanismo de detecção do trabalho de Zhang et al. [25] não suporta novos padrões arquiteturais e variações dos padrões.

B. Describing Software Architecture Styles Using Graph Grammars [17]

Segundo Le Metayer et al. [17] a arquitetura de *software* infere ao sistema atributos de qualidade como: escalabilidade, performance, reusabilidade, dentre outros. Entretanto, ao longo da evolução do projeto podem ocorrer desvios (violações na arquitetura) que geram futuras erosões arquiteturais.

1) Motivação: A principal motivação para o trabalho de Le Metayer [17] está na identificação de ocorrências como desvios arquiteturais. As violações na arquitetura do *software* são muitos comuns em grandes projetos devido a inexistência de uma representação explícita de arquitetura. Os desenvolvedores costumam modificar o sistema sem uma total compreensão sobre a arquitetura prescritiva do *software*.

2) Técnica usada: Para auxiliar no processo de análise arquitetural, Le Metayer [17] propõe uma forma automatizada de avaliar a evolução arquitetural. Nesta forma proposta os estilos arquiteturais são representados utilizando gramáticas de grafos. O *software* a ser analisado é submetido a uma prova de convergência, onde o grafo obtido a partir do sistema deve coincidir com o grafo esperado. Para implementar a prova de convergência, Metayer [17] propõe um algoritmo de verificação que é apresentado em seu trabalho.

3) Ponto Fraco: O trabalho de Le Metayer [17] propõe a representação de estilos arquiteturais usando gramáticas de grafos. Para tanto ele usa como estudo de caso o estilo arquitetural *Client-Server* e outros estilos arquiteturais aplicáveis à sistemas distribuídos não são validados.

C. BGG:A Graph Grammar Approach for Software Architecture Verification and Reconfiguration [7]

O trabalho de Chen Li [7] prover uma gramática de grafo, BGG (*Breeze Graph Grammar*), para descrever a arquitetura do *software* em termos de grafo. Esta gramática de grafo oferece uma base adequada para a reconfiguração da arquitetura e verificação de estilos arquiteturais.

1) Motivação: Segundo Chen Li [7] a arquitetura de *software* fornece uma abstração de alto nível para sistemas de grande porte. Entretanto, a maior parte das técnicas de descrição arquitetural são incapazes de captar mudanças na definição da arquitetura do *software*.

2) Técnica usada: O principal objetivo do trabalho de Chen Li [7] é prover uma gramática de grafo para verificação e reconfiguração da arquitetura de *software*. Segundo Chen Li [7], a maior parte das ADLs (*Architecture Description Languages*) existentes são incapazes de implementar a verificação de estilos arquiteturais e não suporta evoluções na arquitetura do *software*.

3) Ponto Fraco: O trabalho de Chen Li [7] poderia ter apresentado a técnica utilizada para implementação de geração da gramática BGG equivalente ao modelo arquitetural que implementa o estilo *Client-Server*.

D. An Approach of Software Architectural Styles Detection Using Graph Grammar [16]

Este trabalho [16] apresenta a utilização da gramática de grafos ASGG (*Architectural Style Based Graph Grammar*) para a detecção de estilos arquiteturais. Esta abordagem centra-se num modelo arquitetural escrito em xADL, uma das linguagens de descrição da arquitetura mais conhecidas.

1) Motivação: Segundo Songpon [16], grandes empresas frequentemente desenvolvem sistemas de larga escala e alta complexidade. Esses tipos de *software* são difíceis tanto para compreender quanto para detectar estilos arquiteturais. A gramática de grafos ASGG (*Architectural Style Based Graph Grammar*) e suas regras de produção, permite através da detecção revelar algum potencial problema de projeto tal como a não conformidade com algum requisito não-funcional.

2) Técnica usada: Formalizar uma nova gramática de grafos, ASGG, baseada na CS-NCE (*Context Sensitive Graph Grammar with Neighborhood Controlled Embedding*), para detecção de estilos arquiteturais.

3) Ponto Fraco: Um estudo de caso do esquema para detecção de estilos arquiteturais é demonstrado, onde o resultado da gramática de grafos especificada é apresentado e validado. Contudo, assim como no trabalho de Chen Li [7], este poderia ter apresentado a técnica utilizada para implementação de geração da gramática equivalente a um dado modelo arquitetural.

E. Approach to software architecture verification and transformation [18]

O trabalho de Jun Kong et al. [18], propõe uma abordagem para definição e verificação da arquitetura de *software* e suas transformações durante a construção do sistema. Para isso seu desenvolvimento é baseado no uso de gramáticas de grafos para descrever a arquitetura do sistema e através de um analisador de sintaxe, executar a verificação das suas transformações.

1) Motivação: Segundo Jun Kong et al. [18], a arquitetura de *softwares* é geralmente representada por notações UML por serem de fácil compreensão e fáceis de usar. Contudo, notações UML não suportam a verificação de transformações na arquitetura de forma automatizada. Para tratar esta limitação, Jun Kong et al. [18] propõe a utilização das gramáticas de grafos para descrever a arquitetura do sistema, uma vez que as gramáticas permitem um alto nível de abstração sobre a organização geral da arquitetura do *software*, e fornecem uma base formal para verificar a evolução dinâmica de uma arquitetura.

2) Técnica usada: Nesta abordagem, a arquitetura do *software* é descrita por uma gramática de grafos e a verificação arquitetural é executada via analisador de sintaxe.

3) Ponto Fraco: O trabalho propõe uma ferramenta de conversão de modelo arquitetural representado por notação UML, para o formalismo definido pela gramática de grafos RGG (*Reserved Graph Grammar*). Essa ferramenta realiza as transformações para geração do grafo e realiza o parser para verificação das propriedades de qualidade especificadas para o sistema. Entretanto, o analisador de sintaxe é um passo que ainda precisa ser aperfeiçoado para contemplar outros atributos da gramática de grafos adotada pelo projeto e é específico para os atributos da gramática RGG.

Os trabalhos correlatos apresentados nesta seção diferem entre si pelas técnicas usadas, escopo, limitações e objetivos alcançados. Contudo a motivação de todos consiste em minimizar a discrepâncias existentes entre as arquiteturas prescritivas e descritivas do sistema que são responsáveis pela degradação arquitetural do *software*. Em todos os trabalhos correlatos, a atividade de análise arquitetural é destacada pelo seu papel de grande relevância dentro do processo de desenvolvimento do *softwares* e classificada como essencial para a arquitetura

do *software*. Fazendo um comparativo entre estes trabalhos correlatos e o trabalho apresentado neste artigo, pode-se notar que é quase predominante o uso da técnica de gramática de grafos para detecção de estilos ou padrões arquiteturais. O diferencial do trabalho apresentado, comparado com os trabalhos correlatos citados, está na disponibilização de uma ferramenta de detecção que combina gramáticas de grafos com técnicas de *model checking* para detecção de estilos arquiteturais. A ferramenta implementada para detecção serve inclusive de apoio para os demais trabalhos correlatos que utilizam gramáticas de grafos, a exemplo o trabalho de Jun Kong [18] cuja verificação arquitetural é executada via analisador de sintaxe, mas poderia ser executada pela *check graph on full state space*, desde que as gramáticas propostas e representação do sistema estejam modelas no GROOVE. O objetivo do trabalho é o mesmo dos demais trabalhos correlatos: ser uma mecanismo de apoio à análise arquitetural.

VII. Limitações

São vários os estilos arquiteturais classificados na literatura sobre arquitetura de *software* [5]. Para o mecanismo de detecção apresentado neste trabalho, o escopo foi limitado à quatro estilos arquiteturais voltados para sistemas distribuídos: *Client-Server*, *MapReduce*, *Peer-To-Peer* e REST. Para cada estilo arquitetural escolhido, foi implementada um gramática de grafos com suas regras de produção e a avaliação foi feita com base num grafo inicial que representa a arquitetura do sistema modelo, definido para validar a gramática. O escopo do trabalho é limitado por não considerar a detecção de outros estilos, contudo o objetivo principal é fornecer gramáticas de grafos para os estilos verificados e a validação do mecanismo de detecção usando gramáticas de grafos combinadas com técnicas de *model checking* no GROOVE. Uma vez validado que o mecanismo funciona bem, têm-se como contribuição as gramáticas implementadas e o mecanismo de detecção que pode ser estendido para outros estilos arquiteturais, desde que estes tenham suas gramáticas implementadas. Apesar do escopo ser inicialmente reduzido, ele pode ser estendido com novas implementações de gramáticas de grafos sem a necessidade de alteração na lógica do mecanismo de detecção – representado pela funcionalidade *check graph on full state space*, implementada no GROOVE.

O modelo de sistema definido para validar as gramáticas de grafos, contém os principais elementos arquiteturais e restrições presentes num sistemas que implementa determinado estilo arquitetural. Entretanto, nesse modelo não está sendo considerado que pode ser implementado num sistema mais de um estilo arquitetural, o que é comum em aplicações complexas como sistemas distribuídos. Essa limitação deve ser considera para trabalhos futuros, pois as gramáticas de grafos não foram validas nesse cenário.

Além do escopo reduzido de estilos detectados, é importante ressaltar que ao adotar a utilização da técnica de *model checking*, algumas limitações especificas da verificação de modelos foram incorporadas ao trabalho, a exemplo o problema conhecido como explosão do espaço de estados, que é um desafio à aplicação da verificação de modelos. Na simulação do possíveis estados, a depender da complexidade do sistema o registro de todos os comportamentos possíveis de determinado modelo, pode esgotar os recursos de memória da máquina, mesmo que o número de estados alcançados pelo sistema seja finito [23].

VIII. Conclusões e Trabalhos Futuros

Este trabalho apresentou o projeto e desenvolvimento de um mecanismo para detecção automática de estilos arquiteturais voltados para sistemas distribuídos, utilizando gramáticas de grafos e técnicas de *model checking*. O seu principal diferencial está no uso da verificação formal para validar sistemas, descritos por uma linguagem formal, frente a um conjunto de propriedades. A escolha pelo uso das gramáticas de grafos e técnicas de *model checking* ocorreu devido ao fato de serem amplamente aplicadas na ciência da computação para a validação de sistemas e à existência de várias ferramentas que já implementam técnicas de *model checking*, como o GROOVE [24]. O mecanismo de detecção foi implementado no GROOVE, através da implementação da funcionalidade *check graph on full state space* e junto com a criação das gramáticas de grafos representa o diferencial comentado no início deste parágrafo.

As aplicações distribuídas estão cada vez mais presentes no nosso cotidiano (redes de telefonia, sistemas bancários, controle de semáforos, dentre outros), com isso cresce a demanda por métodos que atestem a consistência, completude e corretude do *software*. A complexidade desses tipos de aplicações faz com que seus projetos estejam mais suscetíveis a erros que impactam na conformidade com requisitos funcionais e de desempenho esperados [3] [2]. Ao mesmo tempo, detectar tais erros se torna uma tarefa cada vez mais difícil [25]. Nesse contexto, as técnicas de *model checking* auxiliam que o projetista verifique quando um modelo de sistema satisfaz (ou não) uma dada especificação formal definida no projeto arquitetural [24].

A principal contribuição deste trabalho, está na implementação do mecanismo de detecção no GROOVE e criação com validação das gramáticas de grafos para cada estilos arquitetural escolhido para detecção. O uso das gramáticas de grafos implementadas, combinado com as técnicas de *model checking* e a funcionalidade de detecção implementado no GROOVE, formam um mecanismo para verificação automática de propriedades acerca do sistemas, através de inspeção sistemática no espaço de estados resultantes. A utilização do mecanismo para análise arquitetural referente à sistemas em fase de desenvolvimento, deve permitir a identificação de violações sobre decisões arquiteturais e garantir que a arquitetura do sistema em construção estará de acordo com o projeto arquitetural definido.

Espera-se que o mecanismo permita agilizar a análise arquitetural e auxilie na compreensão da arquitetura e rigor do cumprimento das decisões de projeto evitando a ocorrência de desvios e erosões arquiteturais. É importante destacar que ao detectar a presença dos estilos arquiteturais no sistema, pode ser inferido também a presença dos atributos de qualidades induzidos pelos estilos arquiteturais detectados.

No mecanismo proposto, foi dada enfase à implementação da funcionalidade *check graph on full state space* e criação com validação das gramáticas de grafos para representação dos estilos arquiteturais: *MapReduce*, *Client-Server*, *Peer-To-Peer* e REST. As gramáticas de grafos e a representação

do sistema em linguagem formal, foi realizada no GROOVE que é um projeto de código aberto, escrito em linguagem Java e que já implementa técnicas de *model checking*, porém não realizava a verificação de grafos, fazendo com que fosse necessário a implementação da funcionalidade *check graph on full state space*. Dentre as sugestões de trabalhos futuros, as mais relevantes seguem detalhadas logo abaixo:

- Num primeiro momento foram selecionados alguns dos principais estilos arquiteturais para sistemas distribuídos. Entretanto nada impede que a detecção inclua outros estilos arquiteturais. Nesse caso há a necessidade de um estudo minucioso sobre o estilo a ser acrescentado para a modelagem e implementação da sua gramática de grafos correspondente;
- A ferramenta utilizada para a realização do *model checking* e a implementação da funcionalidade *check graph on full state space* é um projeto de código aberto que pode ser especializado para integração com outras ferramentas. Uma especialização de relevância seria a implementação de integração do GROOVE com o DuSE-MT, ferramenta para modelagem de *software* construído em cima do *framework QtModeling* [26]. O DuSE-MT, viabiliza o processo de determinação da arquitetura do sistema a partir dos seus artefatos de implementação. A saída gerada pelo DuSE-MT para representar a arquitetura do sistema é um XMI (*XML Metadata Interchange*). A integração do GROOVE com o DuSE-MT deverá permitir que o grafo usado para representar o sistema a ser verificado, possa ser gerado a partir de um modelo arquitetural obtido do DuSE-MT e representado por arquivos XMI;
- Outro trabalho de importante relevância consiste na proposta de novas validações para cada uma das gramáticas de grafos modeladas e para o mecanismo de detecção implementado no GROOVE. A validação utilizada neste trabalho, considerou um modelo de sistema com os elementos arquiteturais básicos para determinada implementação de estilo arquitetural. Outras variações podem ser geradas para atestar a completude da gramática de grafos modelada para os estilos arquiteturais detectados e para a validação do mecanismo de detecção implementado. Nesta proposta de trabalhos futuros, cenários mais complexos de modelos de sistemas e novas produções para as gramáticas de grafos seriam criadas visando contemplar alguma variação maior da arquitetura.

REFERÊNCIAS

[1] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992. [Online]. Available: http://doi.acm.org/10.1145/141874.141884

[2] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[3] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[4] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[5] M. Shaw and D. Garland, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[7] C. Li, L. Huang, L. Chen, and C. Yu, "Bgg: A graph grammar approach for software architecture verification and reconfiguration," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on*, July 2013, pp. 291–298.

[8] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang, "A performance study of bittorrent-like peer-to-peer systems," *Selected Areas in Communications, IEEE Journal on*, vol. 25, no. 1, pp. 155–169, Jan 2007.

[9] A. Khan and R. Heckel, "Model-based stochastic simulation of super peer promotion in p2p voip using graph transformation," in *Data Communication Networking (DCNET), 2011 Proceedings of the International Conference on*, July 2011, pp. 1–11.

[10] D. Zhang, C. Zheng, H. Zhang, and H. Yu, "Identification and analysis of skype peer-to-peer traffic," in *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, May 2010, pp. 200–206.

[11] R. Fielding and R. Taylor, "Principled design of the modern web architecture," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000, pp. 407–416.

[12] D. Miner and A. Shook, *MapReduce Design Patterns : [building effective algorithms and analytics for Hadoop and other systems]*. Beijing, Köln, u.a.: O'Reilly, 2013, dEBSZ. [Online]. Available: http://opac.inria.fr/record=b1134500

[13] J.-M. Shih, C.-S. Liao, and R.-S. Chang, "Simplifying mapreduce data processing," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, Dec 2011, pp. 366–370.

[14] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[15] H. Nakada, H. Ogawa, and T. Kudoh, "Stream processing with bigdata: Sss-mapreduce," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, Dec 2012, pp. 618–621.

[16] S. Thongkum and W. Vatanawood, "An approach of software architectural styles detection using graph grammar," *International Journal of Engineering and Technology*, vol. 6, no. 2, pp. 123–127, 2014.

[17] D. Le Metayer, "Describing software architecture styles using graph grammars," *Software Engineering, IEEE Transactions on*, vol. 24, no. 7, pp. 521–533, 1998.

[18] J. Kong, K. Zhang, J. Dong, and G. Song, "A graph grammar approach to software architecture verification and transformation," in *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, Nov 2003, pp. 492–497.

[19] G. Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software," ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2004, vol. 3062, pp. 446–453. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-25959-6_35

[20] J. Rekers and A. Schürr, "Defining and parsing visual languages with layered graph grammars," *JOURNAL OF VISUAL LANGUAGES AND COMPUTING*, vol. 8, pp. 27–55, 1997.

[21] D.-Q. Zhang, K. Zhang, and J. Cao, "A context-sensitive graph grammar formalism for the specification of visual languages." *Comput. J.*, vol. 44, no. 3, pp. 186–200, 2001.

[22] G. R. Andrews, *Concurrent Programming: Principles and Practice*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991.

[23] A. Rensink, "The GROOVE simulator: A tool for state space generation," in *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, ser. Lecture Notes in Computer Science, J. Pfalz, M. Nagl, and B. Böhlen, Eds., vol. 3062. Springer-Verlag, 2004, pp. 479–485.

[24] M. Ben-Ari, A. Pnueli, and Z. Manna, "The temporal logic of branching time," *Acta Informatica*, vol. 20, no. 3, pp. 207–226, 1983. [Online]. Available: http://dx.doi.org/10.1007/BF01257083

[25] L. Zhang, Y. Sun, H. Song, F. Chauvel, and H. Mei, "Detecting architecture erosion by design decision of architectural pattern." in *SEKE*. Knowledge Systems Institute Graduate School, 2011, pp. 758–763.

[26] S. S. Andrade and R. J. d. A. Macêdo, "A search-based approach for architectural design of feedback control concerns in self-adaptive systems," in *Proceedings of the 7th IEEE Intl Conf. on Self-Adaptive and Self-Organizing Systems*. Philadelphia, PA, USA: IEEE, 2013.