IV CONGRESSO DE INICIAÇÃO CIENTÍFICA DA UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

PIBIC/CNPq/UFCG-2006

MONITORAÇÃO E VERIFICAÇÃO AUTOMÁTICA DE SISTEMAS DE SOFTWARE

João Arthur Brunet Monteiro[1], Ana Emília Victor Barbosa[2], Dalton Dario Serey Guerrero[3]

RESUMO

Neste trabalho foi desenvolvida uma ferramenta, intitulada *Design Monitor*, que dá suporte a monitoração e verificação automática da conformidade de código com o modelo de projeto de sistemas concorrentes. Para tanto, uma extensa pesquisa bibliográfica foi efetuada com a finalidade de definirmos uma solução viável para o problema de verificação de programas concorrentes. Manter a conformidade entre o código desenvolvido e o modelo especificado garante aos projetistas que a construção do *software* está de acordo com as especificações por eles expressas. Técnicas para verificação estática foram propostas anteriormente, no entanto, alguns aspectos dos sistemas de *software* só podem ser observados durante a execução do mesmo. Desta maneira, monitorar o sistema e checar se o seu comportamento em tempo de execução está de acordo com as regras especificadas torna-se uma técnica extremamente útil para descobrir e eliminar faltas no código, mantendo assim a sua qualidade e confiabilidade.

Palavras-chave: monitoração, verificação, sistemas concorrentes

AUTOMATED MONITORING AND VERIFICATION OF SOFTWARE

ABSTRACT

This work presents the Design Monitor, a tool that gives support to monitoring concurrent programs and verifying conformance of the code with the specified model. In order to achieve this, an extense bibliography research was done to define the solution for monitoring and verifying concurrent programs. Keeping the conformance of the developed code with the specified model assures to the designers that the software has been building according to the specified rules. Another approaches that use static analysis were proposed before this work. However, some aspects from concurrent software systems only can be observed during its execution. For this reason, monitoring the system execution and checking its behavior have became very useful to discover and remove faults on code, keeping its quality and reliability.

Keywords: monitoring, verification, concurrent systems

INTRODUÇÃO

A utilização de boas práticas no desenvolvimento de *softwares* orientados a objetos (OO) é fundamental para construção de um bom projeto. Contudo, não garantem que o sistema é correto e robusto, fatores que devem ser observados em qualquer *software* de qualidade. O uso cada vez mais difundido de sistemas de *software* para diversos fins na sociedade, juntamente com o seu crescimento em tamanho e complexidade, têm despertado a necessidade por métodos efetivos que proporcionem a verificação de que a

[1] Aluno do Curso de Ciência da Computação, Depto. de Sistemas e Computação, UFCG, Campina Gande , PB, E-mail: jarthur@dsc.ufcg.edu.br
[2] Ciência da Computação, Mestre, Depto. De Sistemas e Computação, UFCG, Campina Grande, PB, E-mail: aesther@dsc.ufcg.edu.br
[3] Ciência da Computação, Prof. Doutor, Depto. de Sistemas e Computação, UFCG, Campina Grande, PB, E-mail: dalton@dsc.ufcg.edu.br

implementação do sistema corresponde à sua especificação. Essa verificação é essencial para que o sistema funcione corretamente.

Mediante este fato, o interesse pela monitoração em tempo de execução de sistemas de *software* tem aumentado. A monitoração permite verificar propriedades relacionadas com o comportamento (mudanças de estados) dos objetos durante a execução do sistema. Esta técnica pode ser utilizada para diversos fins, como *profiling*, análise de performance, otimização do *software* assim como para detectção, diagnóstico e recuperação de faltas (DELGADO, 2004). Um dos principais objetivos da monitoração em tempo de execução é observar o comportamento do software para determinar se este estão de acordo com o comportamento esperado. Enquanto outras técnicas de verificação, como testes, verificação de modelos e prova de teoremas, visam assegurar a corretude universal dos programas, a intenção da monitoração é determinar se o comportamento da execução corrente do programa estão de acordo com o comportamento especificado. O aumento da complexidade e da natureza ambígua dos sistemas de software e os altos custos de testar têm renovado o interesse pela monitoração.

“Um monitor é um sistema que observa o comportamento de um sistema e determina se este está de acordo com o comportamento especificado" (PETERS, 1999). O monitor observa a execução de um sistema e confronta com a especificação de suas propriedade, verificando se o comportamento do software está consistente. Neste contexto, conformidade de código é um termo utilizado para expressar a relação de obediência entre o código implementado e os modelos abstratos que descrevem o sistema de software. Cada vez mais o sucesso de aplicações complexas vem requerendo o uso de técnicas que visam avaliar a conformidade de código, auxiliando na detecção das divergências entre o modelo abstrato e a implementação (SCHMERL, 2006).

Considerando o projeto (design) como o modelo abstrato, esse engloba aspectos estáticos e comportamentais das estruturas que compõe o software. Os aspectos estáticos estão relacionados com a forma como as entidades da arquitetura está organizadas e como estas se comunicam, permitindo a verificação em tempo de compilação. Já os aspectos comportamentais correspondem à dinamicidade do software, ou seja, com a forma de execução.

Este trabalho tem por objetivo a construção de uma solução que permita a verificação automática da conformidade entre o código desenvolvido e o modelo de projeto de sistemas de software. Mais especificamente, a construção de uma ferramenta que monitore e verifique propriedades comportamentais do *software*. A esta ferramenta demos o nome de *Design Monitor*. É importante salientar que o trabalho efetuado é parte integrante de um outro projeto de pesquisa, intitulado como *Design Checker*. Pretendemos com o *Design Monitor*, disponibilizar uma solução para o problema da falta de mecanismos para controlar e medir a qualidade de projetos de software conduzidos usando processos ágeis.

METODOLOGIA

Nesta seção iremos apresentar todo o processo de pesquisa efetuado, bem como os métodos utilizados, para que o projeto fosse concebido.

O projeto foi desenvolvido no Laboratório do Grupo de Métodos Formais (GMF) do Departamento de Sistemas e Computação (DSC) da Universidade Federal de Campina Grande (UFCG). Neste contexto, para a execução do projeto estavam envolvidos, pesquisadores doutores, doutorandos, mestrandos e alunos de graduação. A infra-estrutura atual do laboratório é utilizada para comportar os equipamentos adquiridos com os recursos do projeto, aumentando assim a quantidade de postos de trabalho.

A interação entre os membros envolvidos neste projeto foi realizada através da geração de documentos intermediários e de reuniões periódicas entre os participantes. Além dessas reuniões, seminários foram promovidos com a participação de alunos e demais pesquisadores envolvidos no projeto. Na divisão de responsabilidades, foram estabelecidos indicadores precisos de avaliação. Fizeram parte ainda desta metodologia as definições de ferramentas e meios de comunicação que permitiram uma interação padronizada entre os componentes da equipe do projeto. Nesta sistemática, os trabalhos relacionados foram divididos em atividades. Cada atividade possuía um coordenador, que atuou como elemento que agregava e sinalizava as ações a serem executadas em cada atividade.

O período inicial do projeto foi dedicado exclusivamente aos estudos das técnicas de monitoração e estudo do conceito de modelo de projeto. Além disso, o conceito de conformidade de código foi amplamente investigado. Após o primeiro mês, o foco foi pesquisar sobre as várias maneiras de se expressar regras de projeto. A partir do segundo mês, tendo em vista que já se possuía um conhecimento para isto, a ferramenta *Design Monitor* passou a ser desenvolvida seguindo o processo de desenvolvimento ágil XP (Extreme Programming). Esse processo foi adotado por ser voltado para equipes pequenas, por sua simplicidade na conduta do desenvolvimento do software, por permitir flexibilidade no desenvolvimento das funcionalidades e por obtermos um amplo conhecimento e experiência nessa metodologia.

RESULTADOS E DISCUSSÃO

Tecnologias, Abordagens e Ferramentas estudadas

Modelo de Projeto (Design)

O modelo de projeto baseia-se nos requisitos do software tipicamente estabelecidos em termos relevantes ao domínio do problema, produzindo a descrição de uma abordagem que solucionará aspectos do problema relacionados com o software. Segundo (SWEBOK, 1998) o projeto de software pode ser caracterizado em dois aspectos:

- Projeto Arquitetural (alto nível): descreve como o sistema é decomposto e organizado em componentes e quais as interfaces entre estes componentes;
- Projeto Detalhado: descrição (iterada e hierárquica) dos componentes (da arquitetura) que seja alcançado um nível de detalhe adequado para permitir sua construção.

Conformidade de Código

Conformidade de código é um termo utilizado para expressar a relação de obediência entre o código implementado e os modelos abstratos que descrevem o sistema de software. Cada vez mais o sucesso de aplicações complexas vem requerendo o uso de técnicas que visam avaliar a conformidade de código, auxiliando na detecção das divergências entre o modelo abstrato e a implementação (SCHMERL, 2006). Considerando o projeto (design) como o modelo abstrato, essa engloba aspectos estáticos e comportamentais das estruturas que compõe o software. Os aspectos estáticos estão relacionados com a forma como as entidades da arquitetura está organizadas e como estas se comunicam, permitindo a verificação em tempo de compilação. Já os aspectos comportamentais correspondem à dinamicidade do software, ou seja, com a forma de execução.

Monitoração em tempo de execução

A monitoração em tempo de execução de sistemas de software é utilizada para diversos fins, tais como, capturar o caminho de execução do software, análise de performance, otimização do software assim como a detecção de falhas (DELGADO, 2004). Na literatura existem diversas definições para sistemas de monitoração em tempo de execução. A definição mais adequada ao contexto deste trabalho é que "Sistema de monitoração é aquele que observa o comportamento do software e determina se este está de acordo com o comportamento esperado."

Lógica Linear Temporal

Lógica Temporal Linear (*Linear Temporal Logic* - LTL), introduzida por Pnueli em (PNUELI, 1977), é um formalismo utilizado para a especificação de propriedades de sistemas concorrentes e reativos. Uma propriedade expressa em LTL deve ser satisfeita para todas as execuções de um sistema, considerando a cada momento no tempo apenas um nico futuro. As fórmulas em LTL são capazes de expressar relações de ordem, sem a necessidade de recorrer à noção explícita de tempo.

Autômatos de Büchi

Um autômato de Büchi (BUCHI, 1962) é uma extensão de um autômato de estados finito para entradas infinitas. Autômatos que aceitam palavras infinitas são úteis para especificar o comportamento de sistemas não-determinísticos, tais como sistemas concorrentes e reativos. Cada fórmula LTL tem o seu automato de Büchi correspondente.

Transformando Fórmulas LTL em Autômatos de Büchi

O comportamento esperado sob os pontos de interesse expressos como fórmulas LTL são convertidos para autômatos de Büchi equivalentes através da ferramenta LTL2BA4J. Essa ferramenta é a versão em Java da ferramenta LTL2BA em ANSI C, que permite a conversão das fórmulas escritas em lógica temporal LTL para autômatos de Büchi conforme descrito em (LTL2BA4J, 2006).

Programação Orientada a Aspectos

À medida que aumenta a complexidade dos sistemas de *software* surge a necessidade de melhores técnicas de programação, objetivando organizar e apoiar o processo de desenvolvimento de *software*. Com isso, no âmbito da Ciência da Computação, essas técnicas de programação têm evoluindo desde

construções de baixo nível, como linguagens de máquina, até abordagens de alto nível, como Programação Orientada a Objetos (POO) (ELRAD et al., 2001).

Segundo (GRADECKI et al., 2003), a engenharia de *software* e as linguagens de programação possuem um relacionamento mútuo. Os sistemas são considerados pela maioria dos processos de desenvolvimento de *software* como unidades (módulos) cada vez menores. Enquanto, as linguagens de programação fornecem meios para definir e compor abstrações das unidades do sistema de diversas maneiras para produção do sistema como um todo.

Em meados da década de 70, com o advento de um novo paradigma de programação, a Programação Orientada a Objetos (POO), tivemos grandes avanços no desenvolvimento de software até os dias atuais, sendo atualmente, o paradigma de programação dominante no desenvolvimento de sistemas de software (LADDAD, 2003). Este paradigma possibilitou a construção de sistemas particionados em módulos (classes) que trabalham em conjunto para fornecer funcionalidades específicas de um conjunto de requisitos do sistema com responsabilidades bem definidas, permitindo maiores níveis de reuso e manutenibilidade.

O princípio da separação de interesses (*concerns)* foi introduzido em (DIJKSTRA, 1976), tendo como objetivo dividir o domínio do sistema em partes menores com o intuito de entender melhor cada parte isoladamente. Um interesse (*concern*) é alguma parte do domínio do sistema que se deseja tratar com uma unidade conceitual única. No desenvolvimento de software, um interesse pode ser visto como um requisito funcional ou não funcional de um sistema. De forma geral, os vários interesses do sistema devem ser separados em módulos - modularizados - de acordo com as abstrações do desenvolvimento de software providas por linguagens, métodos e ferramentas. Podemos classificar os interesses de um sistema de software como (LADDAD, 2003).

- **Interesses do negócio** - capturam a funcionalidade central de um módulo, por exemplo, procedimento de quitação de uma compra;
- **Interesses em nível de sistema** - capturam requisitos periféricos, no nível do sistema e que atravessam múltiplos módulos, por exemplo, segurança, logging, persistência.

A Programação Orientada a Aspectos foi proposta como uma técnica que tem por objetivo melhorar o suporte a modularização dos interesses transversais por meio de abstrações que possibiltem a separação e composição destes interesses na construção dos sistemas de software.

AspectJ

A linguagem AspectJ (KICZALES et al., 1997) é uma extensão orientada a aspectos de propósito geral da linguagem Java. Foi criada pela *Xerox Palo Alto Research Center* em 1997 e posteriormente agregada ao projeto *Eclipse* da IBM em 2002. Além dos elementos oferecidos pela POO como classes, métodos, atributos e etc, são acrescentados novos conceitos e construções ao AspectJ, tais como: aspectos (aspects), conjuntos de junção (*point cuts*), pontos de junção (*join points*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*).

Aspects são os elementos básicos dessa abordagem, pois podem alterar a estrutura estática ou dinâmica de um programa. A estrutura estática é alterada adicionando, por meio das declarações inter-tipos, membros (atributos, métodos ou construtores) a uma classe, modificando assim a hierarquia do sistema. Já a alteração numa estrutura dinâmica de um programa ocorre em tempo de execução por meio dos conjuntos de junção, os quais são selecionados por *pointcuts*, e através da adição de comportamentos (adendos) antes ou depois dos pontos de junção (KISELEV). Usamos a alteração da estrutura dinâmica do código para capturar os eventos que ocorrem durante a execução do software.

Visando a supervisão e o acompanhamento dos andamentos das atividades desempenhadas pelo aluno está sendo utilizada a ferramenta *XPlanner*. Trata-se de uma ferramenta *Web* para planejamento e rastreamento de projetos, voltada a metodologias ágeis, mais especificamente a XP (*Extreme Programming*) (BECK & FOWLER, 2001). Dentre outras funcionalidades, *XPlanner* oferece: suporte para cadastro e rastreamento de projetos, iterações, user stories, e tarefas; geração de métricas (velocidade de desenvolvimento da equipe, horas individuais); acréscimo de notas a user stories e tarefas. A escolha de *XPlanner* justifica-se por três motivos principais: é um software livre, é de fácil utilização e oferece recursos para a implantação das práticas definidas no processo elaborado.

Com o objetivo de estabelecer e manter a integridade dos produtos de trabalho durante todo o ciclo de vida do software está sendo utilizado o *Concurrent Versions System* (CVS). CVS é uma ferramenta utilizada para o armazenamento e controle de versões do código. Trata-se de um sistema que permite que grupos de pessoas trabalhem simultaneamente em grupos de arquivos. Para isto, utiliza um repositório central onde são armazenadas as versões mais recentes do software.

Optamos por Linguagem Linear Temporal (LTL) para expressar as regras de projeto por ser uma linguagem simples e que consegue descrever bem regras comportamentais para sistemas concorrentes.

O Verificador de regras foi desenvolvido afim de checar se as regras escritas em LTL não são violadas pelos sistemas monitorados. Seu funcionamento é simples, as regras em LTL são transformadas em automatos de Büchi (BUCHI, 2001) esses por sua vez são alimentados por eventos ocorridos no sistemas monitorado capturados pelos aspectos introduzidos no código do sistema. A partir daí, o estado corrente do automato de Büchi expressará se o sistema monitorado está violando ou não as regras, ou seja, se um evento ocorrido levar a execução do automato para um estado de erro, há uma violação da regra descrita para o sistema, se não, o sistema está se comportando de maneira esperada.

Design Monitor

Nesta seção, apresentaremos a ferramenta denominada *Design Monitor*. Esta foi desenvolvida visando dar suporte ferramental para à técnica de verificação comportamental. Inicialmente, apresentamos uma visão geral da arquitetura definida para o *Design Monitor*. Em seguida, será apresentada as caracteríticas gerais do projeto e da implementação de cada módulo da ferramenta. Para tanto, iremos considerar a monitoração com relação ao comportamento das *threads* durante a execução de um sistema de software *multithreads.*

Visão Geral

A arquitetura da ferramenta *Design Monitor* é ilustrada na Figura 1. O *Design Monitor* foi desenvolvido na linguagem de programação Java e é composto, basicamente, por dois módulos: o módulo de monitoração e o módulo de verificação. A entrada de ambos os módulos são as regras comportamentais. Cada regra comportamental é composta pela definição dos pontos de interesse e a especificação do comportamento desejado como uma fórmula LTL.

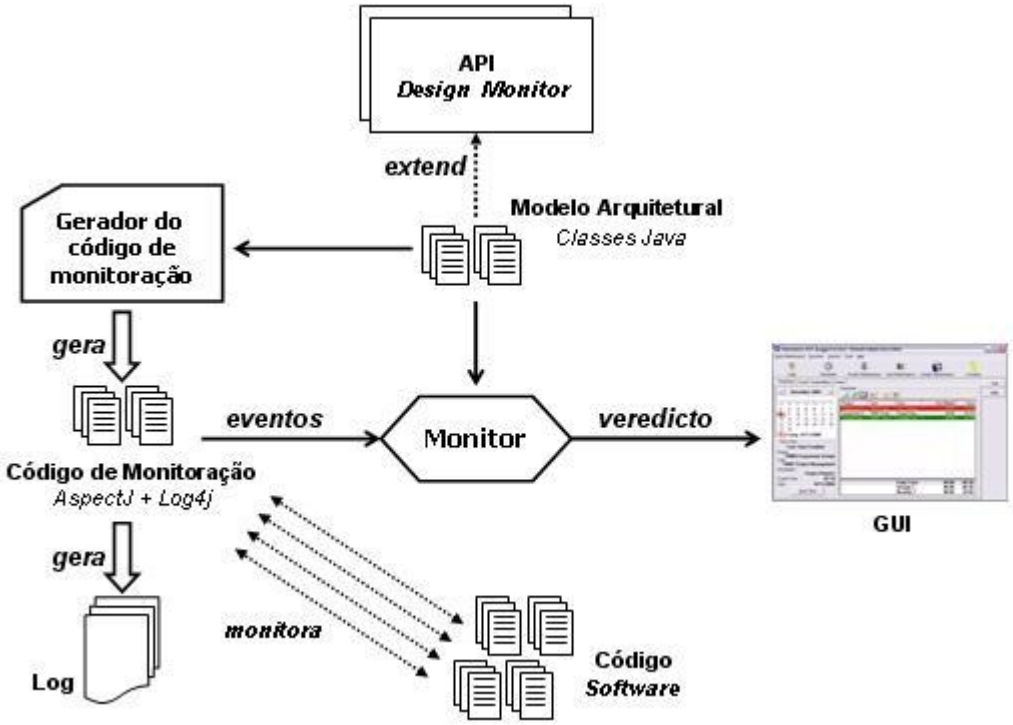

Figura 1. Arquitetura do Design Monitor

O módulo de monitoração é formado pelo código de monitoração, gerado para cada regra comportamental a partir dos pontos de interesse. O código de monitoração é responsável por observar o comportamento do sistema de software nos pontos de interesse durante sua execução. Essa monitoração é realizada através da Programação Orientada a Aspectos (POA). Para tanto, utilizamos AspectJ na implementação do código de monitoração para a linguagem Java. A escolha pelo uso de aspectos se deve ao fato destes permitirem que o sistema de software alvo seja instrumentado de maneira não intrusiva, ou seja, sem alterar o código fonte de origem. Além disso, permite a parametrização com a escolha das classes e/ou métodos a serem monitorados. Os eventos capturados nos pontos de interesse são encaminhados para o módulo de verificação via Log4J (LOG4J, 2006). Log4J é uma API que faz parte do

projeto Jakarta de código aberto (*Open Source*) que tem como objetivo permitir ao desenvolvedor fazer *logging* em suas aplicações). Ela permite ao desenvolvedor controlar, de maneira flexível, cada saída *log* e ativar o *log* em tempo de execução sem modificar os binários da aplicação, esse comportamento pode ser controlado apenas editando um arquivo de configuração. Como características deste *framework* podemos citar a flexibilidade e rapidez de geração de *logging* em tempo de execução, sem inserir custos de performance para a aplicação. Além disso, este permite fazer *logging* remotamente no caso da monitoração seguindo a arquitetura ilustrada na Figura 1.

O módulo de verificação é constituído pelo código de verificação gerado automaticamente a partir das fórmulas LTL. Assim como no módulo de monitoração para cada regra comportamental existe um código de verificação. Esse código de verificação é baseado em autômatos de Büchi, que são obtidos a partir da conversão das fórmulas LTL através do *framework* LTL2BA4J. O comportamento observado nos pontos de interesse para uma determinada regra comportamental que foi enviado pelo módulo de monitoração para o módulo de verificação é analisado pelo código de verificação específico para àquela regra. Caso o comportamento observado viole a regra comportamental o *Design Monitor* comunica ao usuário tal violação.

Como visto, cada regra comportamental gera um código de monitoração e um código de verificação

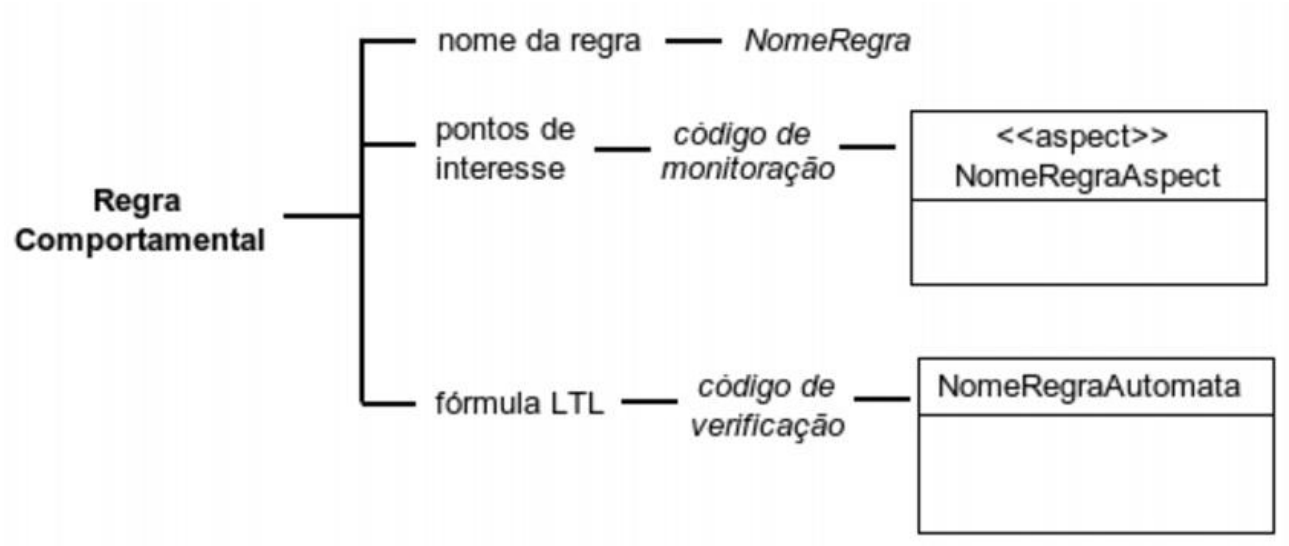

Figura 2. Processo de geração de código no *Design Monitor*.

Implementação

Desde o princípio, o objetivo do projeto da ferramenta *Design Monitor* é permitir que durante o desenvolvimento os desenvolvedores verifiquem automaticamente se o código por eles implementado obedece a certas regras de projeto. Assim, a ferramenta *Design Monitor* foi desenvolvida integrada ao ambiente de desenvolvimento. Para tanto, foi considerado como ambiente a plataforma de desenvolvimento Eclipse.

Ao fazer uso da ferramenta *Design Monitor*, na plataforma Eclipse, temos o projeto do sistema de software a ser monitorado e o projeto da ferramenta *Design Monitor*. No projeto do sistema de software alvo é inserido o código aspecto de monitoração, que foi anteriormente gerado pelo módulo de monitoração do *Design Monitor.* O projeto *Design Monitor* contém o código de verificação também gerado automaticamente no módulo de verificação.

Módulo de Monitoração

O módulo de monitoração é responsável por gerar automaticamente o código de monitoração a partir da especificação das regras comportamentais. O código aspecto de monitoração contém basicamente os pontos de interesse. Para cada regra temos um código aspecto que herda funcionalidades do aspecto DesignMonitorAspect. Basicamente, o código aspecto gerado é conforme o apresentado no Código 1.

```
1 package example.aspect;
2   import ufcg.designmonitor.aspects.DesignMonitorAspect;
3   public aspect NameRule extends DesignMonitorAspect {

5     public NameRule() throws Exception{
6         super("NameRule");
7     }

8     public pointcut monitoringPoints()
9       :   // adiciona os pontos de interesse
10      execution("ponto de interesse 1")
11      && execution("ponto de interesse 2" )
12      ...
13      && execution("ponto de interesse n")
14      && !cflow(execution(int *.hashCode()))
15      && !cflow(execution(java.lang.String *.toString()));
16  }
```

Código 1. DesignMonitorAspect

Quando o sistema de software alvo está sendo executado se um dos pontos de interesse de alguma regra for acessado o código aspecto de monitoração correspondente captura, antes da execução desse ponto (conforme o Código 2 na linha 15), as informações relevantes desse acesso. Essas informações são então organizadas num formato padrão e encaminhadas para o módulo de verificação (Código 2 na linha 12).

```
1 package example.aspect;

2  import java.net.*;
3  import org.aspectj.lang.JoinPoint;
4  import ufcg.designmonitor.log4j.logger.*;

5  public abstract aspect DesignMonitorAspect {

6     protected final static String SEP = "#";
7     protected String nameAspect;

8     public DesignMonitorAspect(String nameAspect) {
9        this.nameAspect = nameAspect;
10    }

11    protected static void trace_record(JoinPoint joinPoint) {
      ...
12      DesignMonitorLogger.getInstance().sendMsg(trace);
13    }

14    public abstract pointcut monitoringPoints();

15    before() : monitoringPoints() {
16       trace_record( thisJoinPoint );
17    }
18 }
```

Código 2 – Código de Abstração de Monitoração

As informações enviadas ao módulo de verificação são as seguintes: o nome da regra cujo ponto de interesse foi acessado; o nome do tipo de objeto acessado, com o método chamado, a linha do método no código de sua classe e o *hashCode* do objeto; o nome da *thread* com seu respectivo *hashCode*; e o *host* onde a informação foi capturada.

Módulo de Verificação

O comportamento esperado sob os pontos de interesse expressos como fórmulas LTL são convertidos para autômatos de Büchi equivalentes através da ferramenta LTL2BA4J. Essa ferramenta é a versão em Java da ferramenta LTL2BA em ANSI C, que permite a conversão das fórmulas escritas em lógica temporal LTL para autômatos de Büchi conforme descrito em (GASTIN, 2001).

O *Design Monitor* é formado por um conjunto de regras comportamentais que devem ser obedecidas pelo sistema de software alvo. Como podemos observar na Figura 3 a classe *Verifier* possui as relações entre as regras comportamentais e os seus respectivos autômatos. Para cada regra comportamental existe uma fórmula LTL, um nome e um conjunto de pontos de interesse (classe *LTLRule*).

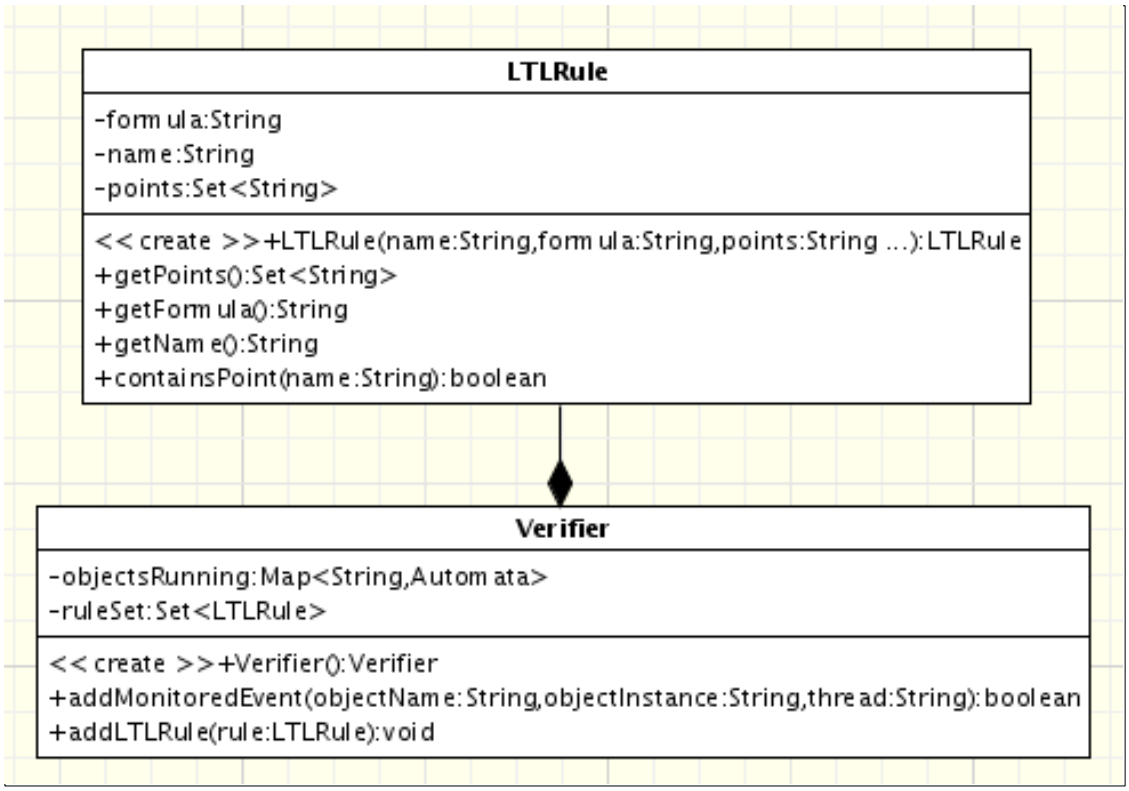

Figura 3. Verificador de eventos observados

As fórmulas LTL são expressas utilizando a mesma sintaxe do verificador de modelos Spin (HOLZMANN, 1997). As fórmulas contém símbolos proposicionais, operadores booleanos, operadores temporais, e parênteses (utilizando espaço entre os símbolos).

Considerando que cada fórmula LTL expressa o comportamento das *threads* sob determinados pontos de interesse, temos que os símbolos proposicionais (proposições atômicas) que compõem essas fórmulas são os nomes das *threads*. As fórmulas LTL são a entrada do módulo de verificação que transforma cada fórmula em um autômato de Büchi equivalente através da ferramenta LTL2BA4J, que tem como saída a representação do autômato de Büchi em um formato de texto. Em seguida, a ferramenta *Design Monitor* converte esse autômato de Büchi gerado numa máquina de estados, cujo código da mesma constitui o código de verificação. Essa máquina de estados deve simular o comportamento das *thread* nos pontos de interesse da sua respectiva regra comportamental. Essa conversão considera que uma fórmula LTL expressa o comportamento desejado, assim todos os estados existentes no autômato de Büchi são tidos como “estados válidos”' durante a execução do sistema. Com isso, não existe o conceito de ``estado final'', o que é existe são os “estados válidos” existentes no autômato fornecido pela ferramenta LTL2BA4J e um novo estado adicionado, denominado “estado de falha”'. Além disso, para cada “estado válido” inserimos uma nova transição entre este estado e o “estado de falha”, cuja função de transição é algum evento diferente das demais transições existentes neste estado.

Na Figura 4 ilustramos através de uma representação gráfica o autômato de Büchi obtido a partir da transformação da fórmula LTL *G(a U G(b))*, onde *a* e *b* são nomes de *threads* existes no sistema de software alvo, que em seguida é convertido na máquina de estado utilizada para acompanhar o comportamento das *threads a* e *b* durante a execução do sistema.

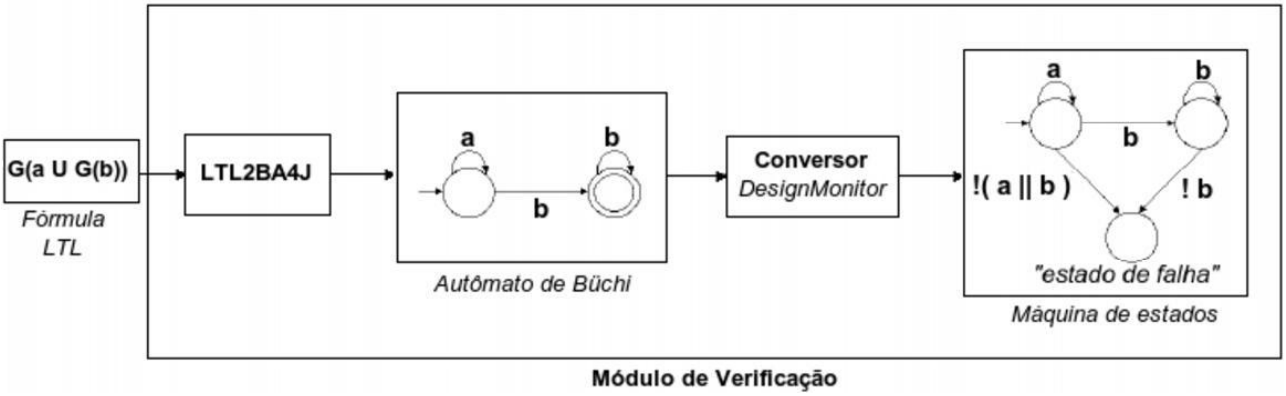

Figura 4. Representação gráfica do processo de conversão na ferramenta Design Monitor

Como pode ser visto na Figura 5 essa conversão é implementada pela classe *Automata*, que recebe como parâmetro em seu contrutor uma fórmula LTL. Durante a execução do sistema de software alvo, para cada regra comportamental é criada um objeto do tipo *Automata*. Suponha a regra comportamental *RuleX*, quando um ponto de interesse de *RuleX* é acessado o módulo de verificação recebe um evento informando o comportamento observado. Então, a máquina de estados referente a regra comportamental *RuleX* executa a operação *run* que recebe como paramêtro o nome da *thread* deste evento. Se este evento levar ao "estado de falha" retornamos *false*, informando que o comportamento observado viola a regra comportamental especificada. Caso contrário, retornamos *true* indicando que o comportamento observado é válido.

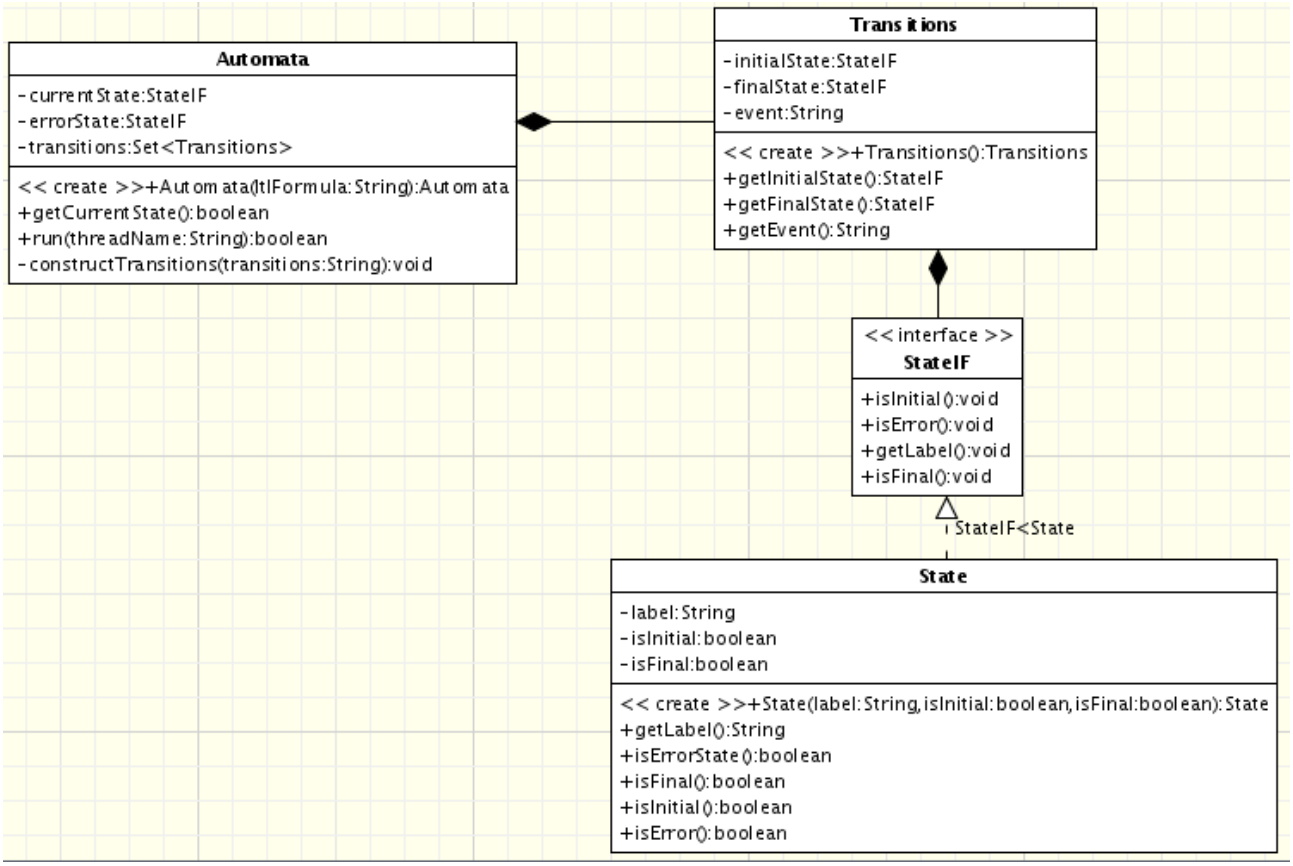

Figura 5. Conversor de fórmulas LTL do Design Monitor

CONCLUSÕES

Manter a conformidade do código desenvolvido com as regras especificadas pelos projetistas é uma técnica que dá suporte a manutenção da confiabilidade do software. Este trabalho propõe uma solução que, analisando aspectos comportamentais do software, verifica se o mesmo está se comportando conforme as regras especificadas previamente. Várias técnicas já foram propostas neste sentido, porém a análise por elas aplicada é estática. Como já discutido neste artigo, alguns aspectos do software só podem ser capturados em tempos de execução. Este cenário se torna mais claro quando se faz uso de concorrência na construção de softwares. A ferramenta *Design Monitor* desenvolvida neste trabalho monitora o software a ser verificado e, analisando eventos ocorridos durante a execução, confronta esses eventos com as regras especificadas através da linguagem LTL.

O uso da ferramenta *Design Monitor* auxilia os desenvolvedores e projetistas na tarefa de detecção de diferenças entre o modelo especificado e o código desenvolvido. Fazendo uso desta técnica, é possível eliminar faltas no código mantendo sua confiabilidade.

AGRADECIMENTOS

Ao CNPq pela bolsa de Iniciação Científica.

REFERÊNCIAS BIBLIOGRÁFICAS

BECK, K. and FOWLER, M. (2001). ***Planning Extreme Programming*.** Addison Wesley.

BUCHI, J. R. (1962). **On a decision method in restricted second-order arithmetic.** In *Proceedings Logic, Methodology and Philosophy of Sciences 1960*. Stanford University Press.

DELGADO, S. M.-N., GATES, M.-A. Q., and ROACH, M.-S. (2004). **A taxonomy and catalog of runtime software-fault monitoring tools.** *IEEE Trans. Softw.Eng.*, 30(12):859–872.

DIJKSTRA, E. W. (1976). ***A Discipline of Programming***. Prentice-Hall. DIJ e 76:1 1.Ex.

ELRAD, T., AKSIT, M., KICZALES, G., LIEBERHERR, K., and OSSHER, H. (2001). **Discussing aspects of aop.** *Commun. ACM*, 44(10):33–38.

GASTIN, P., and ODDOUX, D.. (2001). **Fast LTL to Buchi Automata Translation.** Proceedings of the 13th Conference on Computer Aided Verification.

GRADECKI, J. D., LESIECKI, N. (2003). ***Mastering AspectJ: Aspect-Oriented Programming in Java*.** John Wiley & Sons, Inc., New York, NY, USA.

HOLZMANN, G. J. (1997). **The model checker SPIN.** *Software Engineering*, 23(5):279–295.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., and GRISWORLD, W. G. (2001). **An overview of AspectJ.** *Lecture Notes in Computer Science*, 2072:327–355. Indianapolis: Sams, 2002.

KISELEV, I. (2002). **Aspect-Oriented Programming with AspectJ**.

LADDAD, R. (2003). ***AspectJ in Action: Practical Aspect-Oriented Programming*.** Manning Publications Co., Greenwich, CT, USA.

LOG4J. **Log4J.** Disponível em http://logging.apache.org/log4j/docs/index.html. Acesso em 20 de julho de 2007.

LTL2BA4J. **LTL2BA4J.** Disponível em:
< http://www-i2.informatik.rwth--achen.de/Research/RV/ltl2ba4j/index.html>. Acesso em 20 de julho de 2007.

PETERS, D. K. (1999). **Automated testing of real-time systems.**

PNUELI, A. (1977). **The temporal logic of programs.** *Proceedings 18th IEEE Symposium on Foundations of Computer Science*.

SCHMERL, B. R., ALDRICH, J., GARLAN, D., KAZMAN, R., and YAN, H. (2006). **Discovering architectures from running systems.** *IEEE Trans. Software Eng*, 32(7):454–466.

SWEBOK, A. A., BOURQUE, P., DUPUIS, R., MOORE, J. W. ***Swebok: Guide to the Software Engineering Body of Knowledge****: Trial Version 1.00-May 2001*. Institute of Electrical and Electronics Engineers, October 2002.