# Verificação de Aplicações AmI Usando Java PathFinder

VANDERMI J. SILVA[1], LUCAS C. CORDEIRO[1], VICENTE F. DE LUCENA JÚNIOR[1].
UNIVERSIDADE FEDERAL DO AMAZONAS
AV. RODRIGO OTÁVIO, 3000 - COROADO MANAUS - AM, BRAZIL
E-MAILS: {VANDERMI,LUCASCORDEIRO,VICENTE}@UFAM.EDU.BR

***Abstract*—This paper presents an approach to verify ambient intelligence applications using the model checker Java Pathfinder (JPF). JPF applies the concepts of predicate logic to verify and detect defects in programs, which are not easily detected by conventional testing tools. Testing a software product is not a trivial task, and although there are tools available in the market, there is still the need for automating testing to reduce costs and improve the quality of the software product. The proposed approach shows a scenario that involves the verification of the communication class between the application server and the sensors and actuators, in a completely automated environment, to detect errors related to data race, deadlock, and buffer overflow. The main challenges to check an ambient intelligence application, is the verification of the application layers integrated with the access layer to the hardware sensors and actuators (i.e., access to device drivers). The proposed approach has been applied to verify a real residential automation application, and it was able to detect deadlock problems that were not found by conventional testing environments.**

***Keywords— Automation, Formal Verificationl, Software Test***

## I. INTRODUÇÃO

O uso de sistemas embarcados está cada vez mais presente nos ambientes, sejam eles de escritório, fabril ou residencial, e nesses casos os cenários estão em constantes mudanças, forçando as adaptações e integrações entre diversos produtos, [2] . Com o aumento de aplicativos desenvolvidos para os mais diversos dispositivos, cada vez mais é necessário melhorias nas políticas de verificação de defeitos, que não são facilmente identificados durante a fase de teste, visto que as abordagens convencionais, não garantem que o produto esteja livre de erros como bloqueio fatal, estouro de vetor e corrida de dados, dentre outros problemas analisados por verificadores que são estado da arte, tais como aqueles apresentados em [13], [7],[9] e [10]. Testar um produto de software, não é uma tarefa trivial, e apesar de existirem ferramentas disponíveis no mercado, há a necessidade de automatizar os testes, para melhorar a qualidade do produto e reduzir os custos.

Novas abordagens para verificação de defeitos em software são citados, por exemplo, em [9] e [5]. Essas abordagens permitem verificar formalmente, por meio de modelos matemáticos, como uma aplicação se comporta, possibilitando identificar comportamentos não previstos durante a fase de teste, como por exemplo, bloqueio fatal e corrida de dados.

Os sistemas desenvolvidos para ambientes inteligentes (AmI), possuem como características principais a invisibilidade, a mobilidade, a heterogeneidade, a percepção, a antecipação, a interação com usuário, a observação, o aprendizado e a adaptabilidade [3]. São executados em sistemas embarcados que possuem limitações de memória e processamento e dependendo da aplicação, executam operações críticas, por exemplo, o controle de temperatura de um sistema de calefação em ambientes com baixas temperaturas, necessário para a sobrevivência de pessoas que o habitam.

Esse trabalho apresenta uma importante metodologia para verificação de software para sistemas inteligentes, por meio do framework Java Pathfinder (JPF), uma ferramenta desenvolvida pela Agência Espacial Norte-Americana (NASA), que usa os conceitos de lógica de predicados apresentados em [4] e [17], para verificar e detectar problemas no código que não são facilmente identificados por abordagens de teste convencionais, como por exemplo, testes de caixa branca e caixa preta [15].

No trabalho proposto, foi construído um cenário para verificar um protótipo de software de automação, que coleta dados dos sensores e atuadores, através de uma rede sem fio, para identificar defeitos que causam divisão por zero, corrida de dados, bloqueio fatal e estouro de vetor. Usando arquivos de configuração no ambiente de verificação do JPF e assertivas embutidas no código fonte alvo, foi possível realizar a verificação dinâmica (i.e., execução do software), na aplicação do lado servidor e na aplicação de comunicação entre sensores e atuadores, com o gateway residencial (GR) do ambiente automatizado.

Os principais desafios para verificar um sistema de AmI como um todo, é a verificação das camadas de aplicação integrada com a camada de acesso ao hardware de sensores e atuadores (i.e., acesso aos controladores de dispositivos). O JPF não verifica diretamente tal código nativo. Sendo assim, foi necessário adaptar a verificação do software alvo de forma a iniciar os testes após as chamadas dos métodos nativos, permitindo que os valores de retorno dos métodos fossem verificados. A principal contribuição desse trabalho concentra-se na automação da verificação usando scripts, integrando a compilação e a execução do código dentro do ambiente do JPF, além de dividir as classes testadas para verificar seus métodos isoladamente.

## II. TRABALHOS RELACIONADOS

Nessa seção serão apresentados os trabalhos relacionados às ferramentas de verificação de software, os modelos utilizados na abordagem JPF, e em outras abordagens existentes. Tais trabalhos tratam da verificação formal de software escritos em linguagem de programação Java e C/C++, que servem como base de estudo para o desenvolvimento do trabalho apresentado neste artigo.

Em [13], os autores apresentaram uma abordagem para verificação otimizada de pesquisas em grafos, usando o algoritmo de pesquisa Korat. O experimento foi baseado em duas entradas: uma por meio da implementação da classe Java Predicate e a outra por delimitação da profundidade do grafo a ser pesquisado (bounded). Na implementação do algoritmo, os autores usaram o algoritmo korat, para modificar o bytecode interpretado pelo JPF. Eles descreveram um ambiente de testes, usaram dez estruturas de dados como árvore binária, Heap binário, fila circular, vetores, entre outras estruturas, tendo como resultado a redução do tempo de busca nessas estruturas em 10%.

Em [7], é apresentado um algoritmo de busca, implementado em um framework de testes automatizados para a linguagem Java. O algoritmo usa um método como pré-condição, para gerar automaticamente grafos não isomórficos (i.e., grafos que não possuem nós adjacentes), para os possíveis caminhos da aplicação durante a verificação dos casos de testes. A pós-condição é a saída do teste verificada por meio da lógica de predicados, que como resultado retorna verdadeiro através de uma pesquisa em profundidade, que explora o espaço de entrada do predicado e enumera as entradas para o qual o predicado retorna verdadeiro. Os autores testaram estruturas de dados da linguagem incluindo algumas interfaces do pacote utils, como pilhas, filas e listas da interface Collection do Java, gerando assim casos de testes mais rápidos, se comparados com estruturas de testes baseada em especificações declarativas como visto em [11].

Em [16] é apresentado uma modificação do algoritmo korat, explorando os mesmos espaços de estados do algoritmo, porém considerando os processos em paralelo para cada iteração. Estes processos são distribuídos, resultando em uma versão paralela e eficiente do korat. O algoritmo denominado Pkorat, usa uma configuração mestre e escravo para programar a busca paralela, mas evita que diferentes escravos explorem os mesmos candidatos por meio de meta dados, que evitam a redundância durante a verificação dos nós. Como resultado, o PKorat melhora a velocidade da busca entre 1,9x e 2,6x em relação a implementação do korat.

Em [10] os autores propõem uma abordagem para a verificação semiformal, que combina a verificação estática e dinâmica para verificar sistemas de hardware e software. Os autores desenvolveram uma abordagem que inicia com a criação de casos de teste a partir do sprint backlog. Em seguida, os testes são aplicados no software embarcado e no modelo de processador escrito em verilog, uma linguagem de descrição de hardware usada em sistemas digitais. Após a especificação dos testes e do modelo de processador, o modelo é traduzido para um diagrama de decisão binária, que em seguida é passada para um verificador de modelos. Se o modelo não satisfizer a especificação, um contraexemplo é gerado automaticamente. Esse contra-exemplo é incluído na suíte de testes, e pode ser posteriormente usado para testar o software.

Observando os trabalhos relacionados, percebe-se que, nenhum dos outros trabalhos verificaram especificamente aplicações de AmI, a qual exige a verificação das camadas de aplicação integrada com a camada de acesso ao hardware de sensores e atuadores. Deste modo, a principal contribuição deste artigo concentra-se na proposta de uma abordagem de verificação usando o JPF, para garantir a corretude lógica das funcionalidades que estão tipicamente disponíveis em sistemas de AmI.

## III. FUNDAMENTAÇÃO TEÓRICA

A verificação de modelos (model checking) é uma técnica de verificação de programas para encontrar erros que são difíceis de identificar por meio de testes convencionais de software, como por exemplo, testes de caixa branca e de caixa preta, visto que esses testes levam em consideração somente entradas determinísticas e desta forma, não garantem que todos os possíveis comportamentos do sistema sejam testados de forma a evitar erros tais como, bloqueio fatal ou corrida de dados. A técnica de verificação de modelos consiste em modelar o sistema em uma linguagem matemática, para verificar automaticamente se um conjunto de propriedades é satisfeita durante a verificação, [8]. Tipicamente os sistemas verificados integram hardware, software e protocolos de comunicação, e as especificações exigem requisitos de segurança, como a ausência de bloqueios fatais, estouro de buffer, dentre outros. Para resolver esse problema, o modelo e a especificação são concebidos como uma tarefa de lógica, que checa se uma estrutura satisfaz uma dada fórmula em lógica (proposicional), por meio de uma resposta booleana.

Em linhas gerais, a verificação de modelos com o JPF pode ser descrito da seguinte maneira. Seja M uma estrutura Kripke (i.e., um grafo de transição de estados). Dado φ uma fórmula em lógica temporal (especificação). Encontre todos os s estados de M tal que $M,s \models \varphi$. A partir da estrutura Kripke, o espaço de estados de M são verificados exaustivamente, para determinar se φ (fórmula temporal) retorna uma resposta verdadeira ou falsa. Se a resposta for verdadeira, a estrutura Kripke M é um modelo para a fórmula φ. Caso contrário, um contraexemplo é então produzido, o que demonstra que M não pode satisfazer a fórmula, [8].

Neste trabalho, serão verificadas três classes de defeitos em aplicações de automação residencial. A primeira é o problema da corrida de dados, que consiste em dois processos tentando acessar uma memória compartilhada, sem sincronização apropriada [6]. A segunda propriedade é relacionada ao bloqueio fatal, que é uma situação em que duas ou mais ações simultâneas, esperam a finalização uma da outra para sua continuação, e dessa forma, nenhuma delas pode ser concluída, causando o bloqueio daquele trecho da aplicação. A terceira classe de problema está relacionada ao estouro de vetor, que acontece quando um programa ao escrever dados em uma estrutura de vetor, ultrapassa seus limites e os reescreve em uma região de memória adjacente, violando assim a memória,

[1]. As três condições descritas nesta Seção serão verificadas no estudo de caso apresentado na Seção 5.

## IV. VERIFICAÇÃO DO SISTEMA DE AUTOMAÇÃO COM O JPF

Nesta seção, serão apresentados os passos para construção do ambiente de verificação usando o JPF, iniciando com a metodologia, seguido da configuração da ferramenta, e finalizando com os scripts de execução e ajuste das variáveis.

### *A. Metodologias utilizada*

A metodologia utilizada nesse trabalho consiste em separar as classes que acessam uma base de dados compartilhada, para verificar violações nas estruturas de dados que possam causar estouro de vetor, acessos simultâneos à base que possam resultar no problema da corrida de dados e travamentos, e bloqueio fatal, ocasionados pelo uso incorreto de semáforos.

A metodologia consiste em testar a aplicação como um todo e em separado. A motivação para verificar as classes separadas se deu pela facilidade de testar os métodos das classes isoladamente, para diminuir a quantidade de informações do *log* de erro e para facilitar o controle das classes testadas. A Figura 1 mostra a sequencia de passos utilizados na primeira abordagem. As classes são separadas, e em seguida cria-se uma classe contendo um método principal, que executa os métodos da classe alvo.

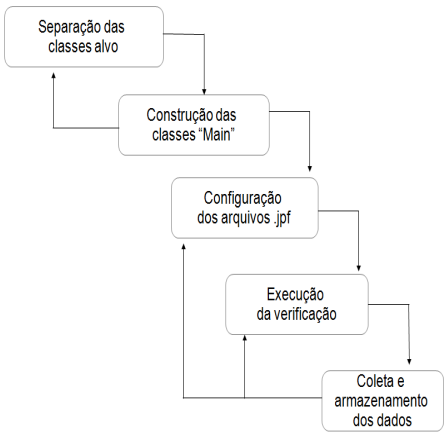


Fig. 1. Sequência de passos utilizados nos experimentos

No próximo passo, os arquivos de configurações *".jpf"* são anexados no ambiente de teste com as propriedades a serem testadas, para em seguida, serem executadas dentro do ambiente JPF, que através de *scripts* de configuração, executam os testes e salvam o resultado em arquivo para posterior avaliação. Na etapa de construção das classes principais, as classes alvo foram instanciadas, e seus métodos testados em uma estrutura de repetição, para simular a chamada automática dos métodos, e coletar os resultados da avaliação. Na etapa de configuração dos arquivos *".jpf"*, cada classe principal recebeu seu próprio arquivo com as configurações de verificação de bloqueio fatal e corrida de dados, além das configurações padrões do JPF que permitem verificar estouro de *buffer* e divisão por zero. Um fragmento de um arquivo *.jpf* e de uma classe *Main* escrito em Java, que foram utilizados neste trabalho, são apresentados na Figura 2. Nesse fragmento pode ser visto a chamada do método *g.lock* e *g.lock.wait* que trava a aplicação enquanto a classe *PesquisaRede* acessa a rede *bluetooth* em busca de novos dispositivos.

Finalizando a metodologia, os resultados dos testes são armazenados em arquivos, que foram tabulados e avaliados. Os resultados da avaliação são apresentados na Seção 6 desse trabalho.

```
Arquivo .jpf
target = TesteCom
listener=gov.nasa.jpf.listener.DeadlockAnalyzer
report.console.property_violation=error,trace
//Classe verificada
public class PesquisaRede {
.....{
        VerificaMac g = new VerificaMac();
        LocalDevice localDevice;
        .......
        } catch (BluetoothStateException e1) {
            .......
          }
//AREA Deadlock
   try {
                synchronized (g.lock) {
                g.lock.wait();
              }
        } catch (InterruptedException e) {
          e.printStackTrace();
    }
```

Fig. 2. Fragmento de um arquivo .jpf e de uma classe com métodos para verificar bloqueio fatal.

### *B. Configuração do ambiente de testes*

A configuração das ferramentas usadas no ambiente de teste foi feita usando um computador com sistema operacional Linux Ubuntu , no qual foi instalado o *Java Development Kit* (JDK) versão 6, o código fonte re-compilado do *framework* JPF, os pacotes de acesso a porta serial/USB, a pilha de *bluetooth* bluecove, e uma API *Extensible Markup Language* (XML) de terceiros para construção da base de dados dos sensores e atuadores da aplicação a ser testada. Além dos softwares, foi necessária também a configuração dos sensores e atuadores que fazem parte do cenário de teste. Para isso foi utilizada uma placa de controle e comunicação de dados provida de entradas/saídas analógicas e digitais, além de possuir também dois relés de contato a seco para acionamento de equipamentos.

A configuração do ambiente consiste em compilar o código fonte do JPF, criar a estrutura de pastas para os testes, e importar as bibliotecas utilizadas no projeto. Após isso, um

*script* de execução dos testes, disponível no projeto JPF, pode ser executado a partir de um terminal do Linux.

Antes de iniciar a verificação com o JPF, é necessário a configuração das variáveis de ambiente, e isso é feito com um *script* a*nt* [12] e [14], que apontam para todas as variáveis de ambiente contidas no projeto, tornando possível o acesso aos comandos diretamente no *console* do computador. Dessa maneira, é possível configurar todo o ambiente de testes, e incluir novas classes para serem verificadas.

## V. Estudo de Caso de um sistema de AmI

Nessa seção será apresentado o estudo de caso de um cenário de AmI usando o hardware e o software apresentados durante a fase de configuração do ambiente de testes na Seção 4. O cenário baseia-se em um controle de acesso a uma residência, por meio de um sistema que reconhece dispositivos *bluetooth* e *wi-fi* previamente cadastrados em uma base XML. O sistema verifica se existem dispositivos na área de cobertura, e se existir, compara o endereço MAC do dispositivo com o armazenado na base de dados.

Se o dispositivo encontrado corresponder ao cadastrado, o sistema envia um sinal via rede *ZigBee* para a placa de controle, que aciona um relé para abrir uma fechadura elétrica. Caso contrário, o sistema mantêm a porta fechada e mostra na saída de vídeo uma mensagem de "entrada não autorizada". Há várias propriedades a serem verificados nesse cenário, e a primeira delas é o problema de corrida de dados, que consiste na concorrência dos processos a uma determinada base compartilhada, [5]. Um dos problemas que podem ocorrer é na consistência dos dados, já que em um dado momento, um processo poderá inserir dados de um novo dispositivo, ao mesmo tempo em que os dados estão sendo verificados por outra aplicação. Nesse caso haverá inconsistência, e o usuário autorizado poderá não ter acesso a residência.

O estouro ou transbordamento de *buffer*, pode ocorrer no sistema avaliado, pois as estruturas utilizadas para armazenar dados em memória, acumulam os *bytes* por um determinado tempo antes de enviarem para a base de dados, e caso haja violação da área de memória disponibilizada, ocorrerá a perda de dados coletados dos sensores.

Outra propriedade verificada é o bloqueio fatal, onde a busca por dispositivos na rede recebe um travamento (*lock*), mas em algum momento o programador esquece-se, após um determinado tempo da pesquisa, de destravar a aplicação (*unlock*). Assim, o método leitor trava todo o processo, e a aplicação fica travada aguardando a finalização da pesquisa na rede.

## VI. Resultados dos experimentos

Essa seção apresenta os resultados dos experimentos usando o JPF para verificação de um sistema de AmI. As classes Java foram testadas separadamente, buscando verificar a corrida de dados e bloqueio fatal. Um exemplo é apresentado na Figura 3. Nesse caso, foi inserida uma área de bloqueio fatal para observar o comportamento do JPF e identificar a falha para ser corrigida pelo programador. O JPF identificou o bloqueio fatal durante a verificação da aplicação, e lançou uma exceção que foi armazenada na base de dados. A Figura 4 apresenta os resultados dos testes com a identificação do bloqueio fatal, enquanto que a Figura 5 mostra o código fonte corrigido usando uma *Thread* em Java, para liberar o processo após o tempo de espera determinado.

```
public class PesquisaRede {
   public PesquisaRede() throws
BluetoothStateException    {
//AREA Deadlock
          try {
                    synchronized (g.lock) {
                    g.lock.wait();
              }
           } catch (InterruptedException e) {
                    e.printStackTrace();
         }
//fim  AREA Deadlock
Arquivo de Configuração .jpf
target = TesteCom
listener=gov.nasa.jpf.listener.DeadlockAnalyzer
report.console.property_violation=error,trace
```

Fig. 3. Código fonte com área de bloqueio fatal e arquivo de configuração.

```
Results    gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:  thread java .lang.Thread:
{id:0,name:main,status:WAITING,priority:5,lockC
ount:1,suspendCount:0}
```

Fig. 4. Relatório de erro com bloqueio fatal identificado.

```
public class PesquisaRede {
   public PesquisaRede() throws
BluetoothStateException    { //outros códigos da
classe.....
//sincronização com bluetooth para evitar
travamento
          //Liberacao em 5 segundos
          try {
                              Thread.sleep(5000);
          } catch (InterruptedException e1) {
                              e1.printStackTrace();
```

Fig. 5. Trecho do código corrigido após a verificação.

Na segunda abordagem, foram testadas todas as classes da aplicação de automação juntas, incluindo as classes de terceiros responsáveis pela busca de endereços de bluetooth e wi-fi, e pelo armazenamento e acesso aos dados em XML. Os resultados desses testes foram parecidos com os resultados dos anteriores, com exceção dos testes das classes de manipulação

do XML, que apresentaram erros de "null point exception" durante a criação do objeto em memória. Para verificar se o erro não era um falso positivo, foi codificado um exemplo da própria biblioteca, com a última versão disponibilizada pelos mantenedores, mas mesmo assim, o erro persistiu. Apesar dos erros encontrados nas classes de terceiros, a aplicação não travou e continuou executando, contudo, isso não quer dizer que não possa acontecer travamentos ou dados inconsistentes, visto que a biblioteca está permitindo instanciar objetos nulos na memória. A Figura 6 mostra o resultado da verificação das classes de terceiro utilizados na aplicação de automação.

```
Gr.java:35 : da.GeraXML_OFF();
Dados.java:87   : XStream xs = new XStream();
[2547 insn w/o sources]
error #1:
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
"java.lang.NullPointerException: Calling
'indexOf(L..."
```

Fig. 6. Erro na instância do objeto XML

Durante a verificação de divisão por zero, estouro de vetor e corrida de dados na aplicação testada, o JPF não identificou tais erros. Para verificar a corrida de dados, foram feitas chamadas em Thread, passando os métodos das classes de acesso aos dados para execução simultânea, com intervalos de acesso de um milissegundo com o intuito de forçar a reprodução do erro no sistema (conforme Figura 7).

```
public class TestDados3 implements Runnable {
                public void run(){
              ....  doSomething(1001);
                        }
        public static void main(String[] args) {
            TestDados3 test = new TestDados3();
                Thread t = new Thread(test);
                        t.start();
                doSomething(1000);
                        }
            static void doSomething (int n) {
                    try { Thread.sleep(n);
                } catch (InterruptedException ix) { }
                        }
}//fim classe
[Lampada: OFF]
java.io.IOException: available() on closed file at
java.io.FileDescriptor.available(gov.nasa.jpf.jvm
.JPF_java_io_FileDescriptor)java.io.FileInp
```

Fig. 7. Verificação da corrida de dados

O resultado foi a concorrência de acesso ao arquivo XML, compartilhado pela Thread que ao tentar acessar o arquivo, gerava uma exceção, pois o processo anterior estava fechando o arquivo após a gravação, gerando assim dados inconsistentes na saída. Após verificar a classe com o JPF, o erro de condição de corrida de dados foi identificado. Ao final dos testes, foi feito a tabulação dos resultados para verificar o consumo de memória, o tempo de verificação das classes, as classes e pacotes verificados, os erros encontrados no código alvo e o número de instruções geradas. Os resultados são apresentados na Tabela 1.

TABELA 1. RESULTADOS OBTIDOS DURANTE A VERIFICAÇÃO COM O JPF

| Classes | LC | T | M | E | TE | I |
|---|---|---|---|---|---|---|
| Pesquisa | 163 | 1,39 | 61 | 1 | *Dead lock* | 2.110716 |
| Armazena Dados | 195 | 0,79 | 61 | 1 | *Null Pointer Exception* | 4.307 |
| Envia CMD | 328 | 0,85 | 61 | 1 | Corrida de Dados | 4.294 |
| Consulta | 241 | 4,01 | 275 | 1 | Corrida de Dados | 14.039.522 |

Os dados coletados durante o experimento e apresentados na Tabela 1 são as classes Java verificadas, o número de linha de códigos implementadas em cada classe dado por LC, o tempo de execução do JPF em milissegundos dado pela coluna T (em segundos), a quantidade de memória RAM consumida durante o processamento do teste, dado por M (em MB), os erros encontrados durante a verificação do código, coluna E, o tipo de erro encontrado, coluna TE, e a quantidade de instruções geradas durante o processamento dos testes, coluna I.

Durante a verificação, foi observado uma condição de bloqueio fatal, na classe PesquisaRede, uma exceção na API de XML na classe ArmazenaDados e duas situações de corrida de dados nas classes EnviaComandos e ConsultaMoradores. A classe que mais consumiu tempo de processamento e memória durante os teste foi a classe ConsultaMoradores, contudo, isso já era previsto, visto que a quantidade de classes herdadas das APIs é bem maior, e como consequência, ocorre o aumento de instruções enviadas para o processador, o que implica em um maior consumo de memória e tempo de processamento. Os erros de corrida de dados e bloqueio fatal foram corrigidos na aplicação de automação que foi utilizada no experimento, entretanto, o erro de null pointer exception ocorrido na API de XML não foi corrigido, tendo em vista que não foi possível acessar o código fonte da API.

## VII. CONCLUSÃO E TRABALHOS FUTUROS

Esse trabalho apresentou uma nova abordagem para verificação de aplicações escritas em Java no domínio de ambientes inteligentes. As classes verificadas durante a execução foram separadas e testadas com o JPF sem a necessidade de *plug-ins* e outros ambientes de programação. Usando scripts integrados no próprio JPF, foi possível portar a aplicação alvo para o ambiente de compilação do JPF, possibilitando automatizar a verificação das propriedades de bloqueio fatal, corrida de dados e estouro de vetor. Durante os testes foram identificados a corrida de dados, condição de bloqueio fatal e um erro em uma API de XML utilizada para acesso aos dados da aplicação. Pretende-se em trabalhos futuros usar essa abordagem para verificar o código fonte de outras APIs Java consolidadas e testadas, além de estender a abordagem para outros domínios de aplicações e integrar o

trabalho com outras ferramentas que verificam código nativo como, por exemplo, as apresentadas em [9], [10].

## AGRADECIMENTOS

Nossos agradecimentos a Fundação de Amparo à Pesquisa do Estado do Amazonas, FAPEAM pelo apoio dado ao projeto por meio de bolsas de estudo, a CAPES e CNPq por meio do apoio financeiro para o desenvolvimento dos projetos.

## REFERÊNCIAS

[1] An Zhiyuan; Liu Haiyan; , "Realization of Buffer Overflow," In IFITA, vol.1, pp.347-349, 2010.

[2] Abowd, G.D., Dey, A.K., Orr, R., Brotherton, J. "Context-Awareness in Wearable and Ubiquitous Computing," in ISWC, pp. 179-180, 1997.

[3] Augusto, R. C.; Carlos, J.; Daniel, S. "Ambient intelligence—the next step for artificial intelligence," in IS, v. 23, n. 2, pp. 15–18, 2008.

[4] Balliu, M.; Dam, M.; Le Guernic, G.; , "ENCoVer: Symbolic Exploration for Information Flow Security," in CSF, pp.30-44, 2012.

[5] Bodden, E.; Havelund, K.; "Aspect-Oriented Race Detection in Java," in TSE, vol.36, no.4, pp.509-527, 2010.

[6] Chang-Seo Park; Sen, K.; Hargrove, P.; Iancu, C.; , "Efficient data race detection for distributed memory parallel programs," in SC, pp.1-12, 2011.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. "Korat: automated testing based on Java predicates," in ISSTA '02, pp. 123-133, 2002.

[8] Clarke E , Grumberg O , Peled D. Model checking. MIT Press, 2000.

[9] Cordeiro, L.; Fischer, B.; Huan Chen; Marques-Silva, J.; "Semiformal Verification of Embedded Software in Medical Devices Considering Stringent Hardware Constraints," in ICESS '09, pp.396-403, 2009.

[10] Cordeiro, L.; Fischer, B.; Marques-Silva, J.; , "SMT-Based Bounded Model Checking for Embedded ANSI-C Software," in TSE, vol.38, no.4, pp.957-974, 2012.

[11] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In Proc. 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, June 2000.

[12] Eric, M. Burke, Jesse Tilly; ,"Ant: The Definitive Guide." O'Reilly Media, May 2002.

[13] Gligoric, M. and et al. , "Optimizing Generation of Object Graphs in Java PathFinder," in ICST, pp.51-60, 2009.

[14] Ken, O. Burtch; "Linux Shell Scripting with Bash." Sams Publishing; 1 edition (February 8, 2004).

[15] Pressman, Roger S.. Software Engineering: A Practitioner's Approach. McGraw-Hill, 6th ed, 2009.

[16] Siddiqui, J.H.; Khurshid, S.; , "PKorat: Parallel Generation of Structurally Complex Test Inputs," in ICST, pp.250-259, 2009.

[17] Tudose, C.; Opria, R.;, "A Method for Testing Software Systems Based on State Design Pattern Using Symbolic Execution," in SEFM, pp 113-117, 2010.