Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

TRABALHO DE CONCLUSÃO DE CURSO

Verificação e Validação de Softwares de Missão Crítica.

Camila Achutti
Graduanda no IME-USP
Orientada

Ana Cristina Vieira de Melo
Livre Docente IME-USP
Orientadora

2 de Dezembro de 2013

Projeto Financiado pela FAPESP (Proc: 2012/16913-2)

Resumo

Esse projeto explora o tema da qualidade de softwares em sistemas críticos através do estudo e aplicação de métodos de verificação e validação sobre um sistema real da área espacial. Nele, um estudo sobre tais técnicas e as principais ferramentas usadas na área foi realizado. Além do estudo sobre os tópicos relacionados à qualidade de sofware, este trabalho se concentrou em um estudo de caso real baseado em um software embarcado espacial, pois a necessidade de verificação e validação é ainda mais intensificada quando se trata de sistemas espaciais. Com isso, o conhecimento teórico obtido foi concretizado na prática através do uso das técnicas sobre um sistema crítico real.

Um estudo sobre a especificação formal do sistema crítico foi realizada para então desenvolver a implementação do projeto na linguagem Java [20, 35] e aplicar técnicas de verificação e validação sobre o sistema desenvolvido. Para tanto, o verificador de modelos *Java Pathfinder* [25], desenvolvido pela NASA foi utilizado já que ele tem sido amplamente desenvolvido pela comunidade científica e se mostrou a melhor opção. Já para validação do sistema foram aplicadas técnicas de elaboração de teste e um ambiente de simulação de eventos espaciais desenvolvido pelo aluno de iniciacão científica Danilo Soares.

Esta monografia foi elaborada para a disciplina MAC 0499 - Trabalho de Formatura Supervisionada e contém duas partes:

- Parte Objetiva, que contém todo os fundamentos teóricos e detalhes técnicos do que foi desenvolvido.
- Parte Subjetiva, que contempla aspectos subjetivos tais como as dificuldades encontradas, como o trabalho se relaciona com o curso e trabalhos futuros.

Palavras-chave: Testes, Verificação Formal, Validação, Sistemas Críticos, JavaPathfinder

Abstract

This project comes to explore the issue of critical systems quality through the study and application of verification and validation methods in a real spatial system. A deep study of such techniques and the main tools used in the area was conducted .

Was also carried out a case study based on an actual embedded spatial software, because the need for verification and validation is further intensified when it comes to this kind of systems .

A study on the critical system formal specification was performed to develop a Java [20, 35] project and apply techniques for verification and validation on this developed system. Thus, the *model checker* *Java Pathfinder* [25] developed by NASA was used since it has been widely developed by the scientific community and proved to be the best option. For system validation, tests were developed and an spatial events simulator were developed by a graduation student, Danilo Soares.

This monograph was prepared for the discipline MAC 0499 Graduation Final Project and contains two parts:

- Objective Part, which contains all the theoretical and technical details of what was developed.
- Subjective Part, which includes subjective aspects such as the project difficulties and challenges.

Keywords : Testing, Formal Verification, Validation, Critical Systems, JavaPathfinder

Sumário

Lista de Figuras

Lista de Tabelas

Parte I

PARTE OBJETIVA

Capítulo 1

Introdução

Dada a ampla utilização de software hoje, seja no auxílio a tarefas simples como escrever uma carta ou tarefas mais complexas como o controle de indústrias químicas, a necessidade de assegurar a qualidade de software torna-se evidente. Uma das formas de assegurar tal qualidade é através de testes do software sobre um conjunto de dados [3, 5, 11, 37, 41]. Contudo, de acordo com Dijkstra, "os testes podem apenas mostrar a presença de erros, não sua ausência" [12]. Diante dessa afirmação, fica claro que apenas testar um software de missão crítica é insuficiente para garantir sua qualidade, o que não significa que a etapa de validação e testes deve ser eliminada, como não será feito no trabalho. Isso já é um fato estabelecido e por isso tem-se hoje investimentos crescentes em métodos de verificação e validação de softwares para garantir níveis de certificação dos mesmos, dependendo da sua criticidade. Essa preocupação se intensifica quando tratamos de softwares embarcados espaciais, uma vez que o investimento na produção desses é enorme e não deixa margem para qualquer falha, afinal, qualquer erro pode colocar projetos complexos em risco e até mesmo ser fatal [6].

Os conceitos de verificação e validação explorados nesse trabalho são os definidos para sistemas computacionais críticos, onde verificação é o processo adotado para determinar que um sistema ou módulo corresponde à sua especificação e, validação é o processo para determinar que um sistema atende seu propósito de funcionamento (em termos de requisitos do usuário). [19]

Uma técnica utilizada para garantir a correção de um software em relação à sua especificação é a verificação formal [4, 7, 17, 33], a qual garante que o software funciona corretamente para quaisquer dados de entrada, diferente dos testes que garantem o funcionamento apenas para os dados específicos. Apesar dos benefícios da verificação formal sobre a qualidade do software serem indubitáveis, essa técnica ainda esbarra em problemas técnico-administrativos para sua aplicação [46]. A aplicação de verificação formal dentro do ciclo de vida do software depende do uso de ferramentas apropriadas para a verificação, as quais, por sua vez, necessitam de treinamento do pessoal que as utilizarão já que envolvem modelagens matemáticas dos problemas. Dessa forma, existe um esforço da comunidade científica para facilitar o entendimento das modelagens formais, assim como para automatizar as tarefas de verificação formal sobre modelos abstratos e sobre programas com o

objetivo de viabilizar o uso dessas técnicas sob o ponto de vista prático.

Dentro desse contexto, nasceram diversas ferramentas para facilitar a realização da verificação e validação, dentre elas nasceu o projeto da NASA do *JPF - Java Pathfinder* [25], que é um verificador (software livre) para programas Java [20, 35]. Hoje, o JPF é conhecido como o canivete suíço para verificação de softwares em Java. De maneira generalista, o *JPF* é uma máquina virtual Java (JVM) que executa um programa não apenas uma vez (como uma JVM normal), mas, teoricamente, de todas as maneiras possíveis, verificando se há violações das propriedades estabelecidas, como *deadlocks* e exceções sem tratamento ao longo de todos os potenciais caminhos de execução. Se encontrar um erro, o *JPF* traça e informa todo o caminho de execução que o levou a esse erro. Esse é o grande diferencial do *JPF* para um depurador normal, ele mantém o registro de cada passo, possibilitando assim a criação desse rastro.

Apesar do *JPF* ser uma ferramenta poderosa para a verificação de programas Java, a sua utilização depende do treinamento de pessoal para definir quais propriedades precisam ser verificadas para um problema em particular e a forma de expressar tais propriedades para que sejam submetidas ao *JPF*. Para definir as propriedades necessárias, o usuário precisa da especificação do problema para decidir quais elementos são críticos e assim determinar as propriedades sobre esses elementos críticos. Uma vez definidas tais propriedades, elas precisam ser descritas em código Java para que sejam submetidas ao JPF. Apesar de existirem propriedades padrão para sistemas em geral [13], a definição das instâncias de propriedades a serem utilizadas por um problema em particular são decisões do projetista e um mesmo projeto pode ter conjuntos distintos de propriedades a serem verificadas. Além disso, não existe atualmente uma metodologia na qual dada uma instância de uma propriedade padrão possamos gerar (manualmente ou automaticamente) uma propriedade a ser submetida no *JPF*.

Na área da validação muito se evoluiu e diversas técnicas de teste e ferramentas foram desenvolvidas e altamente exploradas. No contexto do software de missão crítcia a principal abordagem de teste realizada foi o teste unitário fazendo uso do JUnit, um framework que facilita a criação de código para a automação da execução de testes com apresentação dos resultados. Com ele, cada método de cada classe pode ser validado, testando se funciona da forma esperada, uma vez que expõe possíveis erros ou falhas.

Esse projeto visa estudar um sistema crítico real, desde a sua especificação até a sua implementação, com aplicação de técnicas de verificação e validação de softwares embarcados. Para tanto, a especificação de um sistema espacial será inicialmente estudada e parte do problema que será implementado em Java será isolado, juntamente com os elementos críticos a serem verificados no sistema. Então, novas propriedades para os elementos críticos serão definidas e verificadas no *JPF*.

1.1 Motivação

Existe atualmente uma crescente preocupação mundial com a qualidade dos softwares, sendo ele crítico ou não, já que a evolução dos sistemas computacionais aumentou exponen-

cialmente e, portanto, as funcionalidades implementadas neles são cada vez mais completas e complexas, exigindo que o desenvolvedor e o cliente confiem na correção completa do projeto.

Outro motivo para tamanha preocupação são os grandes investimentos na melhoria dos processos de desenvolvimento de software. Quando tratamos de softwares críticos, essa preocupação é ainda maior. Sendo assim, o investimento em técnicas para validação e verificação de softwares é cada vez maior e imprescindível.

1.2 Objetivo

O objetivo é explorar todos os passos de uma verificação apropriados para um software de missão crítica e validar o software produzido de maneira satisfatória e assim contribuir com a comunidade científica com um estudo sobre verificação formal e validação de programas de sistemas críticos reais.

Capítulo 2

Conceitos e Tecnologias

2.1 Validação de Software Críticos

Validação é aderência aos requisitos desejados do usuário e muito esforço já foi dispendido a fim definir um padrão para a validação dos softwares desenvolvidos para os mais diversos fins, a maioria deles gira em torno da compreensão e acompanhamento dos testes realizados, sendo esses testes os mais variados. No âmbito dos softwares de missão crítica, a principal diferença é que esses testes são dinâmicos, porém realizados fora do seu ambiente real uma vez que isso pode gerar riscos e custos altíssimos e impraticáveis. Sendo assim, os testes devem estar bem estruturados e devem cobrir quase que a totalidade do código a fim de não causarem danos, muitas vezes irreparáveis para o meio em que estão inseridos [19].

Outra prática bastante comum visto o rigor exigido no processo de teste de softwares de missão crítica é elaborado um registro para todos os tipos de teste executados, por funcionalidade testada. Esse documento serve para guiar o processo e também informar a confiabilidade do sistema. As principais técnicas de teste utilizadas a fim de validar um sistema são:

- **Teste Estrutural/Caixa Branca** O teste estrutural tem por objetivo verificar a lógica do programa e a estrutura da codificação. Ele testa a menor unidade de projeto do software, o módulo. Este teste é realizado pelo analista de sistemas, conforme cronograma de desenvolvimento do software. Garantem, assim a solidez e a robustez do software.

- **Teste Funcional/Caixa Preta** O teste funcional visa verificar se o sistema executa as funções especificadas, avaliando assim o comportamento externo do componente de software, sem se considerar o comportamento interno do mesmo. Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido. Como detalhes de implementação não são considerados, os casos de teste são todos derivados da especificação.

- **Teste do Sistema** O teste do sistema visa verificar a disponibilidade de todas as funções especificadas e o atendimento aos requisitos operacionais. Ele é conduzido somente quando todos os módulos estiverem prontos e integrados. São avaliadas todas as funções do sistema. É somente após o período de testes de sistema realizado pelo cliente, pode ser iniciada a geração da versão final do sistema. Este teste somente é realizado para novos softwares desenvolvidos. Softwares existentes que sofreram manutenção ou evolução não passam por este tipo de teste. Nestes casos, é iniciada a geração da versão final do sistema após a realização do teste funcional.

- **Teste de Desempenho** Este teste é não funcional e tem como objetivo avaliar o desempenho do sistema durante sua execução. Ele verifica se o sistema manipula o volume de dados previsto e realiza as transações especificadas dentro de um tempo de resposta adequado a cada função. Este teste deve ser efetuado através de inserção de registros em um volume médio, paralelamente à realização de consultas à base de dados com a finalidade de estressar o sistema e verificar o tempo de resposta obtido. O teste de desempenho será efetuado no próprio ambiente de produção, para todas as operações dos módulos desenvolvidos e devidamente integrados.

- **Teste de Recuperação** O teste de recuperação visa verificar se o sistema se recupera adequadamente de falhas e retoma seu processamento dentro de um tempo pré-especificado em função da criticidade e da importância de cada função. As falhas descritas acima podem ser de diferentes tipos, conforme detalhado a seguir:

  - Falha Humana: Os aplicativos devem evitar que o usuário possa cometer erros como duplicar informações ou excluir indevidamente uma informação;
  - Falha na Transação: Caso ocorram inconsistências no banco de dados, devido a problemas ocorridos durante uma transação de acesso ao mesmo, o mecanismo de restauração rollback do próprio SGBD deve desfazer a operação. Este mecanismo é responsável por desfazer a transação corrente, ou seja, fazendo com que todas as modificações realizadas pela transação sejam rejeitadas

  Esse tipo de teste também é não funcional.

Dentro dessas técnicas podemos depreender diversas fases de testes, sendo as principais descritas abaixo:

- **Teste de unidade** Esse tipo de teste se preocupa com as menores unidades de software desenvolvidas (pequenas partes ou unidades do sistema). Os alvos desse tipo de teste são as subrotinas ou pequenos trechos de código destas. Assim, o objetivo é o de encontrar falhas de funcionamento dentro de uma pequena parte do sistema funcionando independentemente do todo.

- **Teste de Integração** O teste de integração tem por finalidade a avaliação de componentes obtidos pela reunião dos módulos que compõem o sistema. Este teste verifica

principalmente as interfaces existentes entre cada um dos módulos e é realizado pelos programadores no ambiente de desenvolvimento.

- **Teste de aceitação** Geralmente, os testes de aceitação são realizados por um grupo pequeno de usuários finais do sistema, que simulam operações de rotina do sistema de modo a validar se seu comportamento está de acordo com o solicitado e esperado por eles.

2.1.1 Ferramentas e Frameworks existentes

Muitas ferramentas e frameworks foram desenvolvidos a fim de auxiliar as etapas de teste. Além elas são necessárias, pois automatizam algumas etapas das atividades de testes, viabilizando-as na prática. Alguns bons exemplos para possível aplicação no processo de validação de um software crítico são detalhados abaixo:

- Coverlipse [10]: é uma ferramenta para avaliação de cobertura de código. Se tornou um plugin da IDE Eclipse, em que os resultados de cobertura são dados diretamente após a execução dos testes com o JUnit. Suas funcionalidades são:
  - Visualização da cobertura de código dos testes JUnit por meio dos critérios de blocos de instruções , e 'todos os usos';
  - Apenas uma única execução é necessária para a avaliação de todos os critérios de cobertura;
  - Fácil adição/remoção de pacotes do teste;
  - O feedback se dá diretamente no editor da IDE, junto com uma explicação dos resultados obtidos sob um ponto de vista especializado;
  - Não é necessário aprender novos métodos de configuração para a utilização do plugin. Basta configurá-lo do mesmo modo que o JUnit para sua execução normal na IDE Eclipse.
- Poke-Tool [9, 29, 34]: é uma ferramenta desenvolvida na FEEC-Unicamp para apoio ao teste de programas nas linguagens C, Fortran e COBOL. Suas funcionalidades são:
  - Instrumentação do código recebido, viabilizando o rastreamento do caminho de execução dos testes e uma posterior avaliação da adequação de um dado conjunto de teste;
  - Geração dos requisitos de teste (nós, arestas, caminhos ou pares du), feitos de acordo com o critério que o usuário deseja testar. Dentre os critérios disponíveis, estão: todos-nós, todas-arestas e os critérios da família Potenciais-Usos, todos aplicados por meio de uma interface gráfica. Por linha de comando, alguns dos critérios de Rapps e Weyuker [14, 15] para teste de fluxo de dados estão disponíveis, como os critérios todos-usos (em que são considerados todos os usos de

variáveis), todos-c-usos (em que são considerados todos os usos computacionais de variáveis), e todos-usos (em que são considerados todos os usos predicativos de variáveis).

  - Execução e avaliação de casos de teste de acordo com o critério selecionado. O resultado da avaliação é o conjunto de elementos requeridos que restam ser executados para satisfazer o critério e o percentual de cobertura obtido pelo conjunto de teste. Também são fornecidos o conjunto de elementos que foram executados, as entradas e saídas, bem como os caminhos percorridos pelos casos de teste. O processo de execução/avaliação deve continuar até que todos os elementos restantes tenham sido satisfeitos, executados ou detectada a sua não executabilidade..

- EMMA/EclEmma: é um kit de ferramentas escrito em Java para a medição de cobertura de código Java. Pode ser usado tanto em códigos de pequeno porte, quanto códigos de grande porte. Suas principais funcionalidades são:

  - Instrumentação do bytecode do código para análise de cobertura dos testes. A instrumentação pode ser desde arquivos .class individuais até um arquivo .jar completo;
  - Apresentação da cobertura de código de classe, método, linha e blocos, feita por meio da análise da execução de um conjunto de testes unitários, em geral escritos e executados com o auxílio do JUnit. A verificação do EMMA detecta inclusive a cobertura parcial de uma única linha de código;
  - As estatísticas de cobertura são agregadas nos níveis de método, classes, pacotes e "todas as classes";
  - Os resultados da cobertura podem ser exportados nos seguintes tipos de relatório: texto, HTML ou XML;
  - A execução do EMMA não sobrecarrega o sistema, e apresenta os resultados com bastante rapidez.

  Para um uso integrado com a IDE Eclipse, foi desenvolvido o plugin EclEmma, que possui basicamente as mesmas propriedades do EMMA.

- JUnit O framework open-source JUnit é utilizado para auxiliar a escrita de testes unitários em programas Java, e executar o test suite escrito. Ele é amplamente usado pelos desenvolvedores Java, e os testes unitários escritos com ele são focados na verificação das funcionalidades implementadas no código do programa. Dá suporte também para a automatização desses testes. Algumas vantagens de se utilizar JUnit [30]:

  - Agiliza a criação de código de teste enquanto possibilita um aumento na qualidade do sistema sendo desenvolvido e testado;

- Amplamente utilizado pelos desenvolvedores da comunidade código-aberto, possuindo um grande número de exemplos;
- Uma vez escritos, os testes são executados rapidamente sem que, para isso, seja interrompido o processo de desenvolvimento;
- Fornecimento de uma resposta imediata;
- Pode-se criar uma hierarquia de testes que permite a fragmentação do código para realizacão de testes;

2.2 Verificação de Softwares Críticos

Testes são bons instrumentos para detectar erros e, portanto, constituem um processo importante para construir a confiançaa de um programa, tanto para os desenvolvedores quanto para os clientes finais. Entretanto, eles não são capazes de demonstrar que um software está completamente livre de erros: apesar de todo esforço para se construir e selecionar casos de teste que cubram todas as funcionalidades e código do programa, pode haver algum bug que tenha escapado ao processo. Isto ocorre porque os testes não são capazes de demonstrar a ausência de erros no software [12], o que pode se tornar um problema grave no caso de sistemas complexos de grande porte e sistemas críticos, uma vez que uma falha nestes sistemas pode ter efeitos desastrosos. Para suprir esta necessidade de demonstrar a corretude de um programa, surgiu o conceito de verificação de programas. A verificação de programas faz uso de métodos formais para provar que um programa foi implementado corretamente, consiste em conferir se o software está sendo desenvolvido corretamente [8].

De um modo mais formal, podemos definí-la da seguinte maneira: dada uma especificação ϕ e um programa P, mostrar que P é um modelo para a especificação ϕ (i.e., P satisfaz ϕ) [18, 33]. Esta verificação também é conhecida como verificação a posteriori, pois a especificação e o programa são fornecidos. Nela, devemos ter formas de comparar a especificação com o programa para que possamos provar que este último é um modelo para a especifição [33].

Há duas abordagens principais para a realização da verificação de programas: a de verificação de modelos também conhecida como model checking; e a de sistema de prova de programas. Para a realização da verificação de programas, de um modo geral, é preciso seguir algumas etapas [28]:

1. Especificação do Sistema: constitui a base para qualquer atividade de verificação, pois descreve de maneira formal as funcionalidades que devem estar implementadas no programa. Para a realização desta especificação, devem ser utilizados um dos métodos ou linguagens formais disponíveis para o tipo de sistema a ser implementado, de modo a descrever o sistema da forma mais precisa e não ambígua possível [33].

2. Definição de Propriedades: a partir da especificação, deve-se definir propriedades que o produto/protótipo desenvolvido precisa satisfazer. Um exemplo de propriedade, seria a não ocorrência de deadlocks no sistema.

3. Desenvolvimento do Produto/Protótipo: corresponde ao processo de planejamento e implementação do produto/protótipo.

4. Verificação do Programa: com o produto/protótipo pronto para execução, e as propriedades já definidas, é feita a verificação propriamente dita. Caso o produto/protótipo não satisfaça alguma das propriedades de especificação, significa que um erro foi encontrado e, portanto, o programa não está correto. Para fazer esta verificação, podem ser usados verificadores, que são ferramentas que automatizam esta etapa.

Para fazer a verificação de programas, diversas abordagens e técnicas foram desenvolvidas ao longo dos anos. Dentre essas, duas se destacam:

- Verificação de modelos (model checking) [28]: técnica baseada em modelos de sistemas que explora todos os possíveis estados do sistema de maneira exaustiva. O verificador de modelos (*model checker*), ferramenta que faz esta abordagem de verificação, examina todos os possíveis cenários do sistema, de uma maneira sistemática. Desta forma, pode-se mostrar que um modelo de sistema realmente satisfaz uma certa propriedade.

- Sistema de prova de programas [33]: técnica na qual a corretude dos programas é verificada por meio da utilização de um sistema de provas de um dos métodos ou linguagens formais utilizadas para a especificação do sistema, como Z [26], VDM [27], B [47] e lógica de Hoare [21].

A seguir, descrevemos os procedimentos necessários para a aplicação somente da primeira técnica que tem um viés mais prático e é alvo do estudo que esta monografica cobre.

Model checking é uma técnica automatizada que, dado um modelo de estados finitos de um sistema e uma propriedade formal, checa sistematicamente se esta propriedade é satisfeita para um dado estado desse modelo. Esta verificação é feita utilizando força bruta, em que o verificador varre todos os estados do modelo. Caso a propriedade não seja satisfeita, um contra-exemplo é apresentado, mostrando a localização da falha.

As propriedades a serem checadas usando esta técnica são de natureza qualitativa: elas verificam, entre outras coisas, se os resultados gerados são os esperados e se o sistema não entra em situação de deadlock. Outras propriedades que podem ser verificadas são as propriedades de tempo, que podem querer checar, por exemplo, se o tempo de resposta do sistema está dentro do período esperado.

Para aplicar esta técnica de verificação, as seguintes fases devem ser realizadas:

1. Fase de Modelagem:

    - Modelar o sistema em questão usando a linguagem de descrição do modelo do verificador de modelo a ser utilizado;
    - Para uma primeira avaliação do modelo, fazer algumas simulações;
    - Formalizar a propriedade a ser checada usando a linguagem de especificação de propriedades.

2. Fase de Execução: executar o *model checker* para verificar a validade da propriedade no modelo do sistema.

3. Fase de Análise:

   - Se a propriedade foi satisfeita, checar a próxima (se houver);
   - Se a propriedade foi violada, deve-se:
     (a) Analisar o contra-exemplo gerado pela simulação;
     (b) Refinar o modelo, design, ou propriedade;
     (c) Repetir todo o procedimento.
   - Se a memória for insuficiente para a realização da verificação, tentar reduzir o modelo e repetir todo o procedimento.

Ademais, toda a verificação deve ser planejada, administrada e organizada.

Quando estamos falando de sistemas de missão crítica a verificação do programa se faz necessária, mas como toda abordagem, há prós e contra para sua aplicação [28]. Eles devem ser analisados cuidadosamente para que esta técnica seja utilizada da melhor maneira possível, de modo a obter resultados satisfatórios e seja feita de maneira correta e consciente para que seu papel se cumpra.

- Vantagens do model checking:
  - É uma abordagem de verificação geral: é aplicável a uma grande variedade de aplicações, tais como sistemas embarcados, engenharia de software e design de hardware;
  - Dá suporte à verificação parcial: propriedades podem ser checadas individualmente, permitindo assim focar primeiro nas propriedades essenciais. Não é necessária uma especificação de requisitos completa;
  - Provê informação de diagnóstico no caso da propriedade ser invalidada: isto é muito útil para o processo de debugging;
  - O uso de *model checking* não requer um alto grau de interação com o usuário e nem um alto grau de perícia;
  - Goza de um interesse crescente pela indústria;
  - Pode ser facilmente integrado a ciclos de desenvolvimento existentes;
  - Tem suporte matemático: é baseado na teoria de algoritmos de grafos, estrutura de dados e lógica.

  Embora as vantagens da verificação de modelos sejam significativas, deve-se considerar também suas desvantagens. Listamos a seguir alguns dos pontos fracos desta técnica de verificação:

- Sua aplicabilidade é sujeita a questões de ”decisão”para sistemas de estados infinitos, ou raciocínio sobre tipos de dados abstratos (que requer lógica indecidível ou semi-decidível), *model checking* não é eficientemente computável.
- Esta técnica verifica um modelo do sistema, e não o sistema real (produto ou protótipo) em si; qualquer resultado obtido é, portanto, tão bom quanto o modelo de sistema. Técnicas complementares, como testes, são necessárias para ajudar a encontrar erros de código;
- Verifica apenas os requisitos estabelecidos, i.e.,não há garantia de completude. Avalidade de propriedades que não foram verificadas não podem ser julgadas;
- Sofre do problema de explosão de espaço de estados, i.e., o número de estados necessários para modelar o sistema precisamente pode facilmente exceder a quantidade de memória disponível no computador. Apesar do desenvolvimento de diversos métodos efetivos para combater este problema, modelos de sistemas reais podem ser ainda muito grandes para caber na memória;
- Seu uso requer certa perícia em encontrar abstrações apropriadas para obter modelos de sistemas menores e para estabelecer propriedades no formalismo lógico utilizado;
- Não é garantido que ela renderá resultados corretos; como com qualquer ferramenta, o verificador de modelos pode conter defeitos de software, mesmo com testes.

2.2.1 Ferramentas existentes

Para realizar o *model checking*, diversos verificadores e ferramentas têm sido desenvolvidos, de modo a auxiliar na realização de tarefas como a extração de modelos de sistema, escrita das propriedades em linguagem específica, e a realização da verificação em si. Ademais, eles devem lidar adequadamente com os problemas desta abordagem, sendo o principal a explosão de estados do modelo a ser verificado.

Aqui, apresentamos algumas destas ferramentas e verificadores utilizados atualmente para a aplicação da verificação de modelos, mostrando suas principais características.

- SPIN (Simple Promela INterpreter) [23, 43]:
  - Verificador de modelo desenvolvido pela Bell Labs que checa a consistência lógica de uma especificação, e pode ser usado para rastrear erros de design em sistemas distribuídos, tais como sistemas operacionais, protocolos de comunicação de dados e algoritmos concorrentes;
  - O sistema a ser verificado deve estar descrito na linguagem de especificação PROMELA(PROcess MEta LAnguage) [22]; as propriedades, por sua vez, devem ser especificadas em fórmulas LTL (Lógica Temporal Linear) [36, 31, 2];
  - A verificação exaustiva é feita usando a teoria de redução de ordem parcial para otimizar a procura;

  - Trabalha de maneira on-the-fly, o que significa que ele evita a necessidade de se pré construir um grafo de estados global como pré requisito para a verificação das propriedades do sistema.
  - É escalável, tratando eficientemente problemas de grande porte;
  - Oferece suporte ao uso de computadores com múltiplos processadores para execução da verificação de modelos.

- FeaVer (Feature Verification System) [16]:
  - Verificador de modelo desenvolvido pela Bell Labs, usado verificar propriedades no servidor de acesso da Lucent'sPathStar(TM) [45];
  - O modelo do sistema é extraído do código em C e descrito em PROMELA [22]; a verificação ocorre utilizando uma biblioteca de propriedades especificadas em LTL [36, 31, 2];
  - Para a execução da verificação, é utilizado o SPIN [23, 43].

- Bandera [24, 32]:
  - Conjunto de ferramentas desenvolvido pelo Laboratório SAnToS da Universidade do estado do Kansas para a realização de *model checking* em software Java;
  - Dado o código-fonte de um programa Java, retorna o modelo do programa numa das linguagens de entrada dos vários verificadores existentes, como por exemplo a linguagem PROMELA para o verificador SPIN. Bandera também realiza o mapeamento do resultado do verificador para o código-fonte original.
  - Para fazer o processo de tradução da linguagem de programação para a linguagem do verificador, são utilizadas diversas técnicas de otimização como fatiamento, abstração de dados e restrição de componentes;
  - Permite a adição de novas propriedades. Estas devem ser definidas numa linguagem de especificação própria do Bandera, a linguagem BSL [32];
  - Modelos são gerados de acordo com a propriedade a ser testada;
  - Pode ser utilizada de modo integrado a IDEs Java (ex.: Eclipse), e a outras ferramentas de verificação de modelos (ex.: Java PathFinder);
  - Oferece suporte aos elementos introduzidos com Java 5.0 e ao tratamento de exceções.

- JPF (Java PathFinder) [45, 25]:
  - Verificador de modelo desenvolvido pela NASA, que faz uma busca exaustiva de violações de propriedades dado um programa Java;

– Para fazer a busca exaustiva, o JPF foi implementado de modo a funcionar de forma semelhante a uma máquina virtual Java (JVM), na qual o bytecode é analisado e todos os caminhos possíveis do programa são executados;

– As propriedades são implementadas no JPF utilizando a própria linguagem Java e recursos do próprio código de implementação deste verificador;

– É facilmente integrável com outras ferramentas e extensível, além de oferecer suporte aos elementos introduzidos com Java 5.0 e ao tratamento de exceções. Um exemplo de aplicação que estende o JPF é a Java PathFinder Exceptions (JPF Exceptions) [40]: ela checa propriedades de tratamento de exceção num sistema Java utilizando o JPF, e estas propriedades já estáo definidas e implementadas neste verificador. Ademais, ela gera estatísticas de cobertura relativas aos estados varridos durante a verificação.

2.3 O *Java PathFinder* (JPF)

O JPF é o principal objeto de estudo dentro deste projeto. Desenvolvido pelo centro de pesquisas da NASA, começou como um simples verificador de modelos. Ao longo dos anos, este verificador foi aperfeiçoado, tornando-se uma máquina virtual Java que executa um programa alvo de todas as maneiras possíveis, procurando por violações de propriedades como *deadlocks* ou exceções não tratadas. Caso um erro seja encontrado, todos os passos seguidos até a ocorrência do erro são exibidos. Além disso, diferentes extensões foram desenvolvidas para o JPF e estão disponíveis para uso atualmente [25]. Estas variações do JPF incluem, entre outras coisas, implementações de diferentes algoritmos para verificação de programas (ex.: verificação simbólica), criação de pilotos de testes inteligentes e interfaces gráficas para facilitar o uso do JPF e a análise dos resultados da verificação.

Tendo em vista a existência de diversas extensões e aplicações que utilizam o JPF como base, vamos restringir nossa explicação ao núcleo do JPF. Como dito anteriormente, este núcleo é uma máquina virtual, implementada em Java, para *bytecode* Java que dá suporte à verificação de modelos, ou seja, temos um máquina virtual (JPF) executando em cima de outra máquina virtual (JVM). O que foi dito acima é ilustrado na Figura 2.1.

Para utilizar o JPF, o usuário só precisa fornecer como entrada as propriedades a serem verificadas e o programa alvo. Em seguida, o JPF passa a executar o programa recebido como entrada de forma sistemática, em busca de violações das propriedades fornecidas pelo usuário. Aqui, é importante salientar que o JPF apresenta duas características importantes que auxiliam nesta verificação:

- ***Backtracking*:** o JPF retorna aos passos anteriores da execução do programa, para verificar se há estados não executados. Caso tais estados existam, estes serão os próximos a serem executados.

- ***State Matching*:** o estado do sistema é armazenado como um par de uma amostra do *heap* e da pilha da *thread* do programa em execução. Quando um estado já visitado

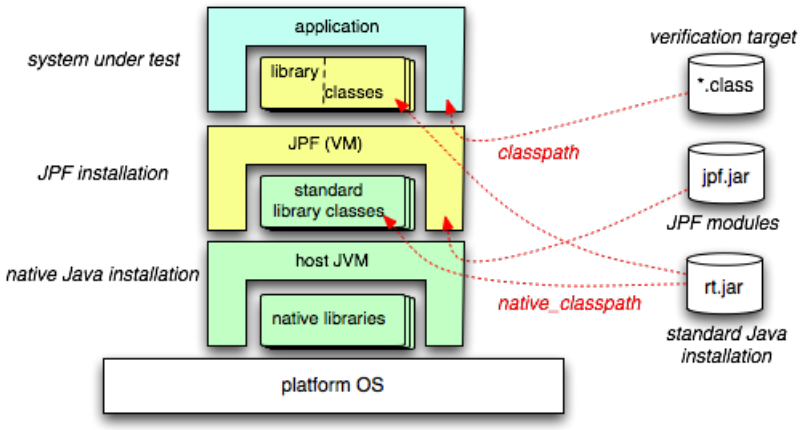

Figura 2.1: Esquema geral do JPF retirado de babelfish.arc.nasa.gov com permissão dos autores.

é encontrado, o JPF realiza o *backtracking* até que um estado novo inexplorado seja encontrado.

Ademais, cabe lembrar que o principal problema a se contornar para viabilizar essa verificação é que o número de caminhos possíveis e, consequentemente, o número de estados a serem executados pode crescer exponencialmente. Este problema, conhecido como ”explosão de estados”, é minimizado pelo JPF com o uso dos seguintes recursos:

1. **Estratégias de busca configuráveis:** consiste na aplicação de filtros que determinam que estados devem ser verificados ou não.
2. **Redução do número de estados:** para este recurso, a ferramenta utiliza os mecanismos listados a seguir:
   - **Geradores de escolha heurísticos:** consiste em restringir as escolhas de valores de teste para um estado.
   - **Redução de ordem parcial:** consiste em considerar somente operações que podem ter efeitos em outras *threads* do programa, tais como instruções que alteram objetos que podem ser acessados externamente. Com a aplicação deste mecanismo, operações com efeito puramente local são desconsideradas, reduzindo assim o número de estados a serem percorridos durante a verificação.
   - **Execução de VM aninhada:** o JPF executa sobre a JVM nativa, o que permite que determinadas operações sejam delegadas para a JVM nativa ao invés de serem executadas pelo JPF. Desta forma, estes estados deixam de ser executados pelo JPF, otimizando a verificação.

Ao fim do processo de verificação, o JPF cria um relatório que diz quais propriedades estão asseguradas: caso alguma tenha sido violada, ele apresenta o trecho do código do

programa analisado que violou a propriedade, de modo que o usuário possa identificar o problema e corrigí-lo.

2.3.1 Mecanismos de Extensão do JPF

Para dar uma maior flexibilidade à ferramenta, mecanismos para a extensão do verificador JPF foram disponibilizados pelos seus desenvolvedores. Dentre os mecanismos, os principais são:

- ***Search/VM Listeners*:** permitem a criação de classes Java que são notificadas quando determinados eventos ocorrem, como por exemplo a execução de determinadas instruções de *bytecode* ou avanço/retrocesso nos passos de execução. Os *listeners* podem ser usados para implementação e verificação de propriedades complexas, execução de buscas, extensão do modelo de estado interno do JPF, ou para colher estatísticas de execução do programa.

- ***Model Java Interface*(MJI):** interface que permite trabalhar no nível do *bytecode* controlado pelo JVM nativa, i.e., permite trabalhar num nível abaixo ao do *bytecode* controlado pelo JPF. Alguns de seus principais usos seriam a interceptação de métodos nativos Java que podem interferir na execução da verificação do JPF, e a possibilidade da execução de VM aninhada, conforme descrito na seção anterior.

2.3.2 Limitações do JPF

Embora o JPF tenha inúmeras funções e seja bastante utilizado no meio científico, ele apresenta algumas limitações que acabam por restringir o seu uso em sistemas reais grandes e complexos. As principais limitações são:

- **Explosão de estados:** apesar dos esforços na otimização do processo de verificação, o problema da explosão de estados continua restringindo a aplicação do JPF em programas de maior porte. A situação se agrava no caso do sistema trabalhar com várias *threads* concorrentemente, o que faz com o que o número de estados a serem percorridos durante a verificação cresça exponencialmente. Isto acaba resultando em demora no tempo de execução da verificação e, em casos mais extremos, na não conclusão da verificação devido ao consumo de memória maior do que o suportado pelo *heap* da JVM nativa. Alguns trabalhos que buscam melhorar este aspecto do JPF seriam as extensões: *jpf-concurrent*, voltada para programas concorrentes [44] e *SPF (Symbolic Pathfinder)*, que implementa a verificação simbólica no JPF [42, 38]. Particularmente, o *SPF* tem apresentado bons resultados em relação à verificação de sistemas de maior porte, sendo usado pela empresa Fujitsu para testes de aplicações web [25].

- **Bibliotecas Java sem suporte nativo no JPF:** algumas bibliotecas Java, como *java.awt*, *java.net* e *java.io* não são suportadas pela versão principal do JPF. Há

algumas extensões do JPF que estão sendo desenvolvidas para oferecer suporte a estas bibliotecas, como o *jpf-awt*, que visa à aplicação do JPF em códigos que utilizam as bibliotecas de interface gráfica do Java; e *net-iocache*, que visa à aplicação do JPF em programas que fazem uso de entrada/saída e comunicações de rede, como por exemplo, códigos de servidores.

Uma outra limitação do uso efetivo do JPF está relacionada à falta de treinamento de pessoas da indústria para a identificação e definição de propriedades a serem verificadas. Este projeto visa contribuir com o treinamento da aluna de Iniciação Científica sobre um sistema crítico real, o qual será usado como estudo de caso para avaliar o uso efetivo do JPF dentro do ciclo de vida de desenvolvimento de sistemas críticos. O resultado desta avaliação será uma contribuição para a comunidade de verificação formal, já que a maioria dos verificadores de modelos têm sido aplicados a pequenos problemas e, em sua maioria, problemas não industriais.

2.4 Ferramentas Uilizadas no Projeto

Para a validação do sistema em estudo, o JUnit foi escolhido pela facilidade da escrita de testes automatizados. Além disso inclui algumas funcionalidades como:

- Asserções para testar resultados esperados
- Fixtures: para reutilização de dados para teste
- Test Suites: para organizar e executar conjuntos de testes
- Interface gráfica e textual para execução de testes
- Integração com as principais IDEs
- Grande comunidade de usuários

Diante disso, o JUnit se mostrou um ótimo framework para auxiliar à tarefa de validação de um sistema crítico que requer tanto cuidado.

Além disso, neste trabalho, optamos por trabalhar com o verificador JPF por ela se diferenciar das demais pelos seguintes motivos:

- Linguagem Java: neste trabalho, o uso de uma linguagem Java foi determinado pelo crescente uso do Java para missões críticas, esse interesse começou na NASA e só a ausência de uma biblioteca totalmente previsível (previsível no sentido de que todas as operações na biblioteca têm uma duração de execução conhecida) impediam que Java fosse utilizada, sendo que normalmente as escolham recaíam sobre C, C++ e Ada. Esse não é mais o caso, como demonstrado por um paper da AIAA(American Institute of Aeronautics and Astronautics), que apresenta a primeira biblioteca totalmente determinística (e open-souce) para Java: Javolution [1].

- Facilidade na especificação de novas propriedades e/ou configuração das implementadas na distribuição padrão: este é um dos pontos mais importantes que diferenciam a ferramenta e o verificador escolhidos das demais aplicações: no caso deles, as propriedades podem ser especificadas utilizando a própria linguagem Java, sem a necessidade de aprender linguagens novas e específicas para o verificador, como ocorre no caso do Bandera e do SPIN.

Além disso, é uma ferramenta open-source com uma comunidade bastante ativa.

Capítulo 3

Desnvolvimento do SWPDC-Java

Para concretização do estudo do caso desse trabalho de conclusão de curso era necessária a escolha de um software de missão crítica real que apresentasse todos os desafios de se trabalhar com um softwares críticos, além disso precisávamos de uma especificação completa para que a etapa de implementação, validação e verificação pudessem se concretizar de forma realista e consistente.

Nesse contexto nasceu o SWPDC [39] é um software real embarcado para gerenciamento de dados espaciais. Ele foi desenvolvido no contexto do projeto de Qualidade de Software Embarcado em Aplicações Espaciais (QSEE) do Instituto Nacional de Pesquisas Espaciais (INPE), sendo assim contava com uma documentação vasta e precisa. Com base na Especificação Técnica do Software era possível ter noção da complexidade do sistema e todos os requisitos e funcionalidades que lhe cabiam.

Além disso, é de interesse do INPE que o sistema seja desenvolvido em Java para que acompanhe os novos rumos da pesquisa espacial e desenvolvimento de softwares de missão crítica. Sendo assim, o SWPDC se mostrou um sistema adequado para o desenvolvimento do trabalho.

3.1 SWPDC

O SWPDC [39] foi desenvolvido no contexto do projeto de Qualidade de Software Embarcado em Aplicações Espaciais (QSEE). Para seu desenvolvimento uma longa e criteriosa documentação foi desenvolvida. Com base na Especificação Técnica do Software produzida é possível ter uma visão completa do funcionamento do sistema, dos seus requisitos e do ambiente de desenvolvimento. A especificação do projeto possui 97 requisitos básicos do sistema relativos aos componentes do processo, distribuídos entre os seus quatro modos de operação (Figura 3.1).

Além disso, a modelagem comportamental de cada um dos modos de operação do SWPDC também é detalhada na especificação. Todos os requisitos estão descritos em linguagem natural e foram posteriormente modelados em *Statecharts* [39]. Um estudo detalhado do SWPDC foi realizado, primeiramente para identificar partes específicas do

software a serem efetivamente implementadas, e posteriormente para a correta implementação do sistema.

3.1.1 Detalhes da Especificação do SWPDC

O SWPDC é um software embarcado que já foi implementado em C e Assembly (hardware dedicado). O software possui quatro modos de operação do *Payload Data Handling Computer* (PDC), sendo eles: Iniciação, Segurança, Nominal e Diagnóstico. Tem, ainda, 5 processos componentes, sendo eles: Comunicação, Controle de Serviços de Aplicação, Gerenciamento de Dados, Gerenciamento de Estado e Suporte. A relação existente entre esse modos de operação do PDC e os componentes do SWPDC estão ilustrados na Figura 3.1.

Processos Componentes do SWPDC	Modos de Operação do PDC			
	Iniciação	Segurança	Nominal	Diagnóstico
Comunicação		■	■	■
Controle de Serviços de Aplicação		■	■	■
Gerenciamento de Dados		■	■	■
Gerenciamento de Estado		■	■	■
Suporte	■	■	■	■

Figura 3.1: Relação entre os modos de operação do PDC e os componentes do SWPDC.

O processo de Comunicação citado acima é responsável por realizar a comunicação entre o *On-Board Data Handling Computer* (OBDH) e os *Event Pre-Processors* (EPPs). Já o Controle de Serviços de Aplicações distribui os comandos recebidos do OBDH para os processos destinatários. O Gerenciamento de Dados é responsável pelo gerenciamento de memória com os 5 tipos de dados existentes (científicos, diagnóstico, teste, descarga de memória e *housekeeping*), pela formatação dos dados para telemetria, pela carga e execução de programas na memória do PDC e pela descarga da memória do PDC. O processo de Suporte realiza funções de inicialização do SWPDC/PDC (são nessas funções que o bom funcionamento de hardware é testado) e aquisicão e atuação em canais discretos e no hardware do PDC. Por fim, o Gerenciamento de Estado realiza o gerenciamento do modo de operação do PDC, inclusive o processo de troca de modo de operação, assim como implementa mecanismos de tolerância a falhas no sistema, que deverão detectar,

confinar e recuperar a ocorrência de erros, além disso, deve gerar relatórios de eventos que os causaram.

Dependendo do modo de operação, o SWPDC deve dar suporte também aos serviços mostrados na Figura 3.2.

Serviços suportados pelo SWPDC	Modos de Operação do PDC			
	Iniciação	Segurança	Nominal	Diagnóstico
Obtenção/Formatação/ Transmissão de Dados Científicos			■	
Obtenção/Formatação/ Transmissão de Dados de Diagnóstico				■
Obtenção/Formatação/ Transmissão de Dados de Testes		■		■
Obtenção/Formatação/ Transmissão de Dados de Descarga de Memória			■	■
Obtenção/Formatação/ Transmissão de Dados de Housekeeping		■	■	■
Carregamento e Execução de Programação			■	

Figura 3.2: Relação entre os modos de operação do PDC e os tipos de serviço do SWPDC.

3.2 SWPDC desenvolvido

Grande parcela do trabalho dispendido para a produção desse trabalho foram aplicados no estudo dos 2 principais documentos, cedidos pelo INPE, relacionados ao software SWPDC. Sendo eles os seguintes:

- Especificação Técnica do software - QOO-ETS-V03
- Documento do Projeto do Software - Arquitetura POO - QOO-DPSW-V05

Diante do estudo dos mesmos, para simular o funcionamento do software, o desenvolvimento de dois produtos independentes, um é o sistema SWPDC (Software Piloto Embarcado no Payload Data Handling Computer) propriamente dito, implementado em Java e nas especificações descritas nos documentos citados acima. Tal sistema preparado para rodar e obter dados do segundo produto totalmente independentemente para simular o PDC (Payload Data Handling Computer), que na sua essência é um simulador de eventos espaciais e os provê para o software. O segundo produto foi desenvolvido a fim de

tornar a validação do SWPDC ainda mais consistênte e realista. A especificação do mesmo encontra-se anexada a este trabalho.

Sendo assim eles rodam em paralelo conforme a figura abaixo ilustra:

Figura 3.3: Esquema dos produtos propostos

Além disso, como fruto do estudo desses documentos foi possível perceber duas abordagens diferentes feitas pelo sistema em questão que seram resumidamente contextualizadas abaixo:

- Fragmentação por Modos de Operação: mapeia o sistema de um ponto de vista mais subjetivo pensando nos possíveis estados do sistema. Distinguem-se quatro modos de operação, sendo eles:
  - Diagnóstico: responsável pela verificação da "saúde"do sistema como um todo, verificando tanto se os sensores estão funcionando corretamente quanto se surgiu algum erro no software;
  - Nominal: modo de operação de digestão dos dados obtidos pelos sensores, realização de cálculos e construção dos relatórios (produtos finais do sistema)
  - Segurança: momento de recebimento dos dados externos e também momento de isolamento e resolução de problemas advindos de possíveis bugs encontrados.
  - Inicialização: primeiro estado assumido pelo sistema, responsável pela inicialização de todos os componentes e dá início aos primeiros procedimentos do sistema.
- Fragmentação por Componentes: traz uma abordagem mais baixo nível e tenta isolar componentes fortemente dependentes, visando obter uma estrutura modularizada que

faz bastante sentido nos softwares de missão crítica. Tal abordagem reconhece cinco componentes distintos:

- Dados: responsável por todo gerenciamento de dados, desde seu armazenamento até sua formatação;
- Suporte: responsável por procedimentos como a inicialização de sensores e aquisição de dados dos mesmos;
- Estado: gerencia os modos de operação, toda tolerância a falhas e se necessário é o responsável por modificações nos paramêtros do software;
- Comunicação: atua no gerenciamento de todos os canais de comunicação entre os OBDH (On-Board Data Handling) e os EPPs (Event Pre-Processor);
- Serviços: responsável pela execução dos comandos recebidos da interface de comunicação com o OBDH chamando os componentes envolvidos na tarefa.

O maior objetivo desse estudo era identificar com quais desses fragmentos poderíamos começar a trabalhar, levando em conta também, quais produziriam um resultado mais satisfatório, sabendo que seria impossível implementar o sistema inteiro, que é bastante complexo e foi produto de diversos pesquisadores durante anos.

Observando os modos de operação, percebe-se que escolher um deles para analisar e implementar não seria a melhor maneira de começar e de obter um feedback rápido, pois um modo de operação envolve pequenas partes de todos os componentes do software que estariam incompletos e não produziriam o efeito desejado. Sendo assim, optar pelo desenvolvimento por componentes se mostrou a decisão mais adequada, pois assim teríamos um módulo completo implementado e independente do ponto-de-vista de funcionamento. Este aspecto é bastante interesante para a realização da verificação de maneira que essa não tenha que ser refeita neste módulo posteriormente com a implementação de novas funcionalidades do sistema.

Dentre os componentes, a atenção maior foi dada pelo de dados foi a opção feita por ser um dos principais componentes do sistema, já que é responsável pelo gerenciamento de dados, a qual é a funcionalidade central do SWPDC.

Com o conhecimento e decisões tomadas nessa fase, partimos para a modelagem do sistema e foi realizada a implementação, tarefas que serão detalhadas na próxima seção.

3.2.1 Modelagem e implementação do sistema

O produto desta fase está ilustrado no UML simplificado do sistema (Figura 3.4). Foi a partir dessa estrutura inicial apresentada na Figura 3.4 que o detalhamento de cada uma das classes e a efetiva implementação das mesmas, foi feito de maneira progressiva e incremental para obtermos objetos concretos de forma rápida. A primeira rodada de detalhamento e implementações foi das classes relacionadas ao gerenciamento de dados que se valiam de buffers simples no processo.

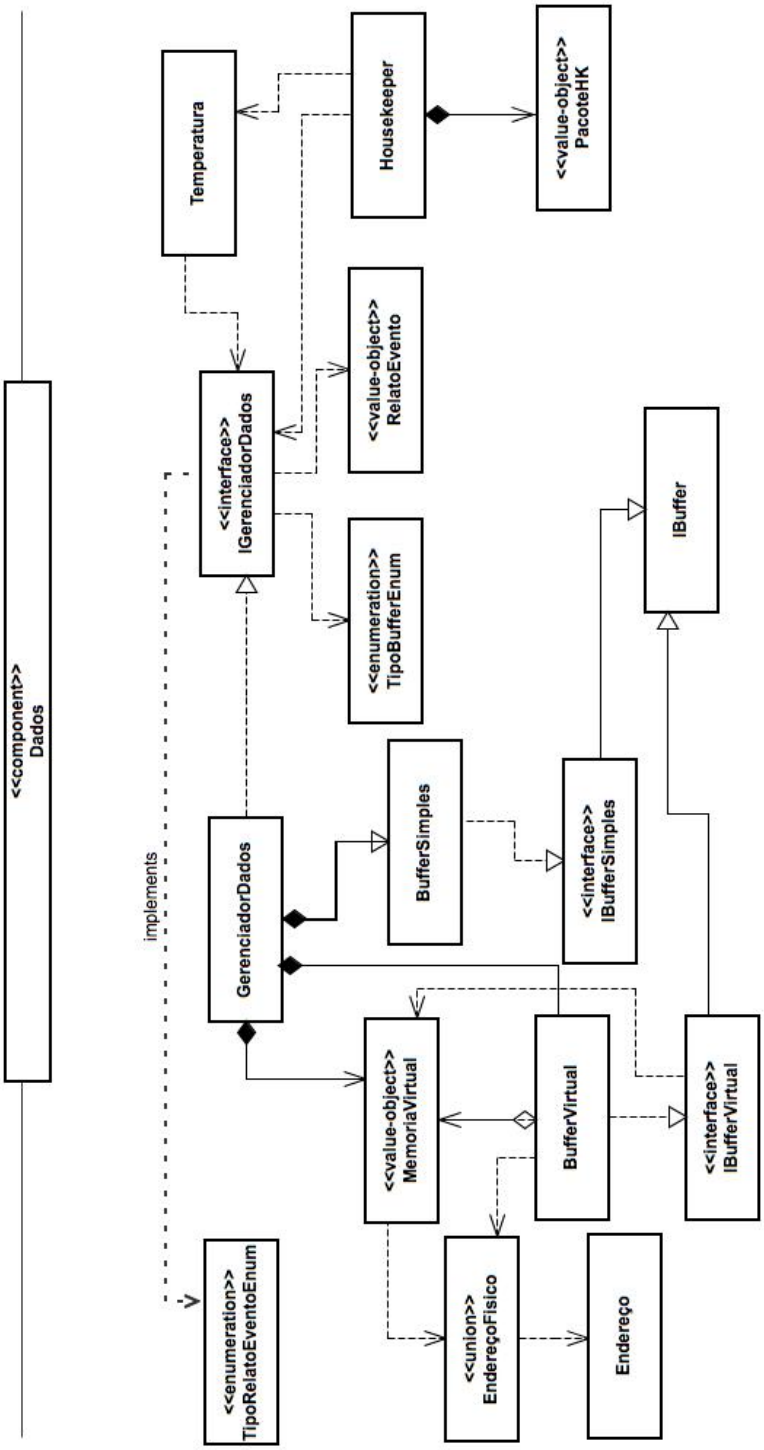

Figura 3.4: Modelagem do componente de Dados

Para implementação das mesmas usamos a biblioteca ByteBuffer que foi anteriormente testada para ver se o *JPF* conseguiria lidar com as funções usadas conforme será descrito mais detalhadamente em seção posterior deste trabalho. O teste viabilizou o uso da biblioteca.

Com o avanço na implementação, tivemos que aumentar o escopo do sistema implementando também o componente de suporte, que tem sua estrutura e interação com o restante do sistema ilustrados nas Figuras 3.5 e 3.6.

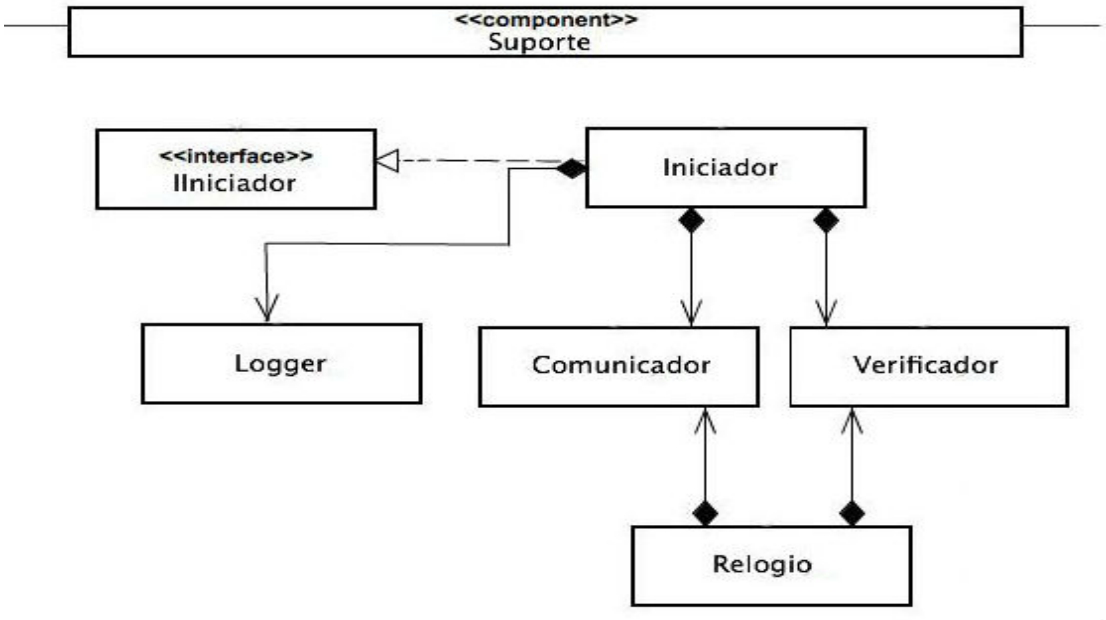

Figura 3.5: Estrutura do componente Suporte

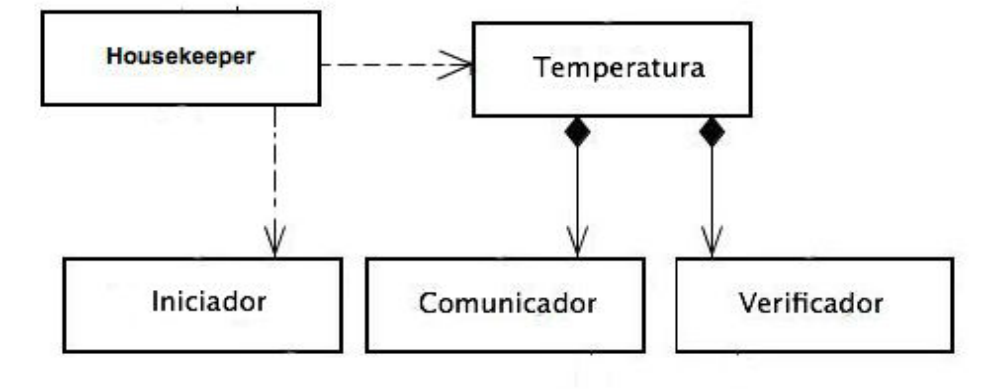

Figura 3.6: Relação entre o componente de Dados e Suporte

Com a independência do simulador e do SWPDC foi necessário modelar também a

comunicação entre o simulador e o SWPDC. As possibilidades levantadas para relizar esse fluxo de dados foram:

- Comunicação por sockets
- RMI (Remote Method Invocation) é uma interface de programação que permite a execução de chamadas remotas no estilo RPC (Remote Procedure Call) em aplicações desenvolvidas em Java. É uma das abordagens da plataforma Java para prover as funcionalidades de uma plataforma de objetos distribuídos.
- Web service é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis.
- Compartilhamento de arquivos

A decisão tomada foi realizar a comunicação por aquivos planos em uma diretorio conhecido e acessivel por ambos, pois no contexto de software embarcado e crítico buscamos um método eficiente, leve e seguro de realizar essa comunicação, sendo assim a maneira mais adequada foi realizá-la via compartilhamento de arquivos, onde temos o ganho de facilidade e persistência leve (ocupa pouco espaço).

Um programa Java abre um arquivo instanciando, criando e associando um objeto ao fluxo de dados. A principal classe utilizada para criar arquivos é a `File` e programas Java implementam o processamento de arquivos utilizando as classes do pacote `java.io.`, para lê-los e editá-los foi feito o uso das classes: `java.io.BufferedReader` e `java.io.BufferedWriter`, respectivamente.

O modelo implementado de fluxo de dados está representado na Figura 3.7:

Os arquivos propostos contêm as seguintes informações:

- `log.txt` (PDC_simulador) - histórico dos erros do PDC_simulador
- `sysInfo.txt` - histórico do estado e hora do simulador. Ocorre uma atualização toda vez que o SWPDC emite o comando: OBTER RELOGIO e/ou INFORMAR MODO DE OPERACAO
- `registroTemp.txt` - histórico de registro de temperaturas. Ocorre uma atualização toda vez que o SWPDC emite o comando: OBTER TEMPERATURAS
- `instrucoes.txt`: troca de comandos. É atualizado pelo SWPDC e monitorado pelo Simulador.
- `relatorioPacoteHK.txt`: resultado do SWPDC. Relatório do sistema.
- `historico.txt`: mantém um track de todo o comportamento do SWPDC
- `log.txt` (SWPDC): log do SWPDC

Para que fosse possível o entendimento do arquivo sem grande overhead de interpretação foram fixados modelos rígidos, principalmente para os arquivos compartilhados. O modelo de cada um dos arquivos encontra-se no Anexo **??** deste trabalho.

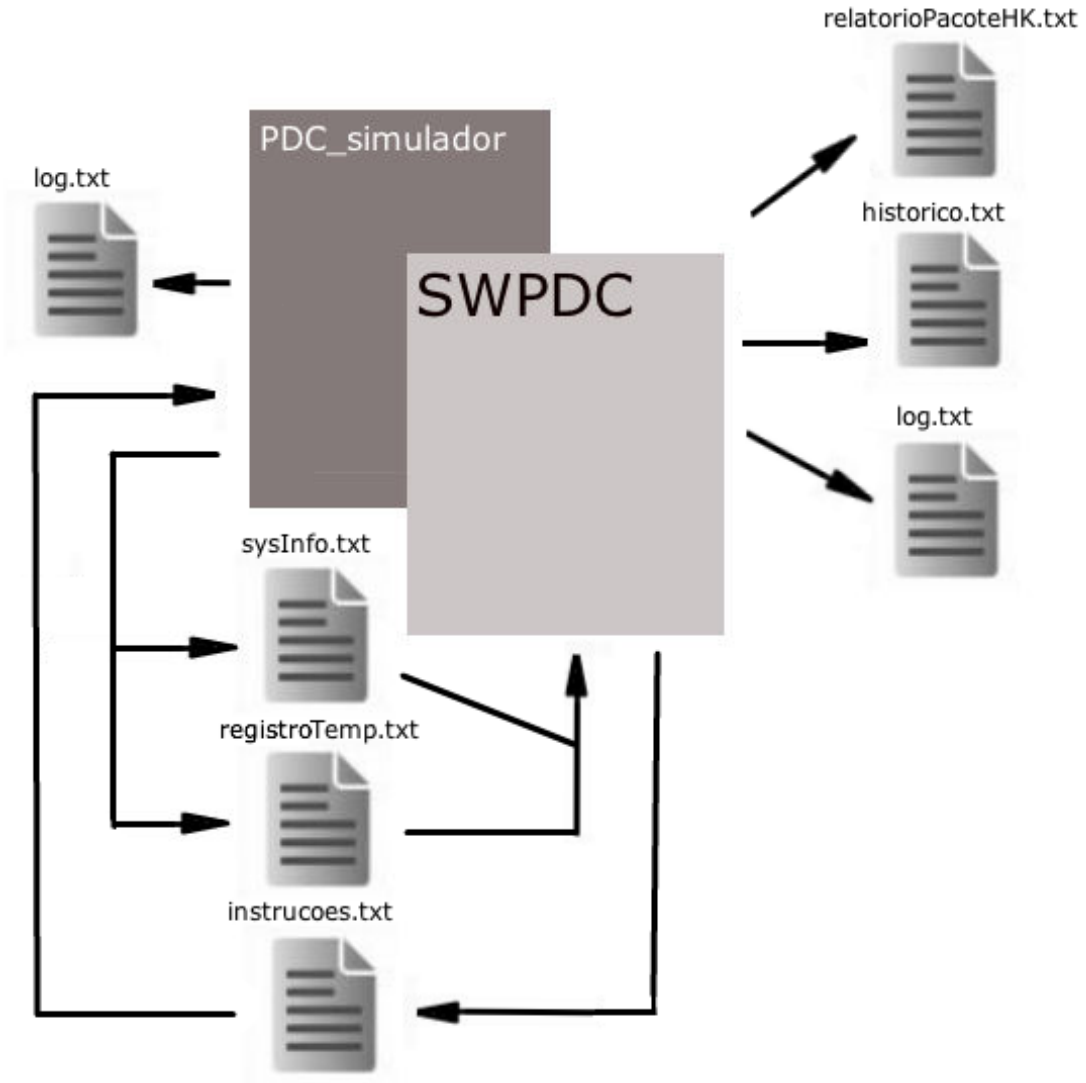

Figura 3.7: Modelagem do fluxo de Dados

Principais Interfaces

- `IBuffer.java`

```
1 package dados;
2
```

```
3  /**
4   *
5   * @author Camila Achutti
6   */
7  public interface IBuffer {
8
9      //Verifica se o buffer esta vazio. 1 indica que o
10     // buffer esta vazio, 0, caso contrario
11     public int estaVazio();
12
13     // Insere um dado no buffer, dado - Ponteiro para
14     // endereco do primeiro byte do dado a ser inserido
15     // no buffer, tamanho - Tamanho do dado a ser inserido
16     // no buffer. Retorno - A quantidade de dados
17     //efetivamente gravada no buffer.
18     public byte inserir(int dado, byte tamanho);
19
20     // Remove dados do buffer, dado - Ponteiro para area
21     //de destino dos dados a serem removidos do buffer,
22     // tamanho - Quantidade de bytes a serem removidos.
23     // Retorno - A quantidade de bytes efetivamente
24     // removida do buffer.
25     public byte remover(int dado, byte tamanho);
26
27     //Retorna o tamanho do buffer em bytes.
28     public byte tamanho();
29 }
```

- `IBufferSimples.java`

```
1  package dados;
2
3  /**
4   *
5   * @author Camila Achutti
6   */
7
8  //Interface para buffers que manipulam memoria de area
9  // comum a todas paginas (memoria estatica, nao paginada).
10 public interface IBufferSimples {
11
12     //Registra a alocacao de uma area de memoria para
13     //ser manipulada pelo buffer.
14     //area - Ponteiro para o inicio da memoria alocada
15     //tamanho - Tamanho da area alocada a ser registrada.
16     public void alocar( Object area, byte tamanho);
17
18
19 }
```

- `IGerenciadorDados.java`

```
1  package dados;
2
3  /**
4   *
5   * @author Camila Achutti
6   */
7  public interface IGerenciadorDados {
8
9      // Adiciona dados a um buffer especifico,
10     //destino - Indicador do buffer de destino,
11     // o qual recebera os dados, dado - Ponteiro
12     // para o inicio da area a ser copiada
13     //(inserida) no buffer, tamanho - A quantidade
14     // de bytes a ser inserida no buffer.
15     //Retorno - A quantidade de bytes efetivamente
16     // inserida no buffer.
17     public double adicionar( TipoBufferEnum destino,
18                                      int dado, byte tamanho);
19
20     // Altera ponteiros do buffer de descarga de
21     //memoria. endInicial - Endereco Inicial,
22     //endFinal - Endereco Final.
23    public void altBufDmp(double endInicial, double endFinal);
24
25     //Altera endereco inicial do buffer de descarga.
26     //endInicial - Endereco Inicial.
27     public void altEnderecoInicialDmp(double endInicial);
28
29     //Altera selecao de memoria que se pretende fazer dump.
30     //mem - Indicador da memoria a ser selecionada.
31     public void altSelecaoMemDmp(int mem);
32
33     //Efetua os procedimentos de iniciacao do gerenciador
34     // de dados.
35     public void iniciar();
36
37     // Efetua os procedimentos de iniciacao do
38     //buffer de carga.
39     //endereco - Endereco inicial para carga.
40     public void iniciarCga(double endereco);
41
42     //Efetua os procedimentos de iniciacao do buffer
43     // de diagnostico da camera HXI 2.
44     public void iniciarDgeHx2();
45
46     //Efetua os procedimentos de iniciacao do buffer
47     // de descarga de memoria.
48     public void iniciarDmp();
49
50     //Efetua os procedimentos de iniciacao do buffer
51     // de teste.
```

```
52      public void iniciarTst();
53
54      //Obtem endereco inicial do buffer de descarga.
55      // retorno - O valor do endereco inicial
56      //do buffer de descarga.
57      public double obtEnderecoInicialDmp();
58
59      //Obtem selecao de memoria que se pretende
60      //fazer dump. retorno - O valor do indicador
61      // da memoria selecionada.
62      public int obtSelecaoMemDmp();
63
64      //Obtem quantidade de pacotes ainda disponiveis.
65      //subtipo - Subtipo dos dados de resposta.
66      // retorno - O valor do CSR (Controle de
67      //Sequencia de Resposta) corrente.
68      public double obterCSR(int subtipo);
69
70      // Retorna ultimo indicador de pulling dos
71      //conjuntos EPP Hi - HXi (idHX).
72      //retorno - Valor do indicador idHXi.
73      public int obterUltimoIdHx();
74
75      //Insere no buffer de relato de eventos um dado evento,
76      // juntamente com sua informacao associada,
77      //tipo - Indicador do tipo do relato de evento.
78      //info - Informacao associada ao evento.
79      //Deve ser obrigatoriamente um vetor de 5 bytes.
80      public void relatarEvento( TipoRelatoEventoEnum tipo,
81                                                  int[] info);
82
83      //Remove dados de um buffer especifico,
84      // local - Indicador do buffer de
85      // origem dos dados a serem removidos,
86      // dado - Ponteiro para a area de
87      // memoria de destino, o qual serao copiados
88      // os bytes removidos do buffer.
89      // tamanho - Quantidade de bytes a serem removidos
90      // do buffer. retorno - A quantidade de bytes
91      // efetivamente removida do buffer.
92      public double remover( TipoBufferEnum local, int dado,
93                                                 byte tamanho);
94
95
96  }
```

Principais Classes

- `BufferSimples.java`

```java
package dados;

import java.nio.ByteBuffer;
import java.util.NoSuchElementException;
/**
 *
 * @author Camila Achutti
 */
//buffer circular
public class BufferSimples implements IBufferSimples,
                                                    IBuffer {

    private Object m_area; //"Ponteiro" para a area de
                        // dados alocada para este buffer.
    private int m_ixrd; //indice leitura do buffer.head
    private int m_ixwr; //indice de escrita do buffer.tail
    private byte m_tamanho; //Tamanho, em bytes, da area
                            //de dados alocada
    ByteBuffer buffer;

    public BufferSimples (Object area, byte tamanho){
       m_tamanho = tamanho;
        m_area = area;
        buffer = ByteBuffer.allocateDirect(tamanho);
        m_ixwr = buffer.limit();
        m_ixrd = buffer.position();
       alocar (area, tamanho );
    }

    //Registra alocacao
    @Override
    public final void alocar(Object area, byte tamanho){
      //e no caso de nao ter rolado a alocacao correta
        System.out.println("Alocação no Buffer de tamanho"
                                    + tamanho + "registrada");
    }

    @Override
    public boolean estaVazio() {
        if (m_ixwr == 0) {
            return true;
        }

        return false;
    }


    @Override
    public byte inserir(float dado, byte tamanho){

        byte capacidadeInicial;
```

```
52          byte capacidadeFinal;
53          byte bytesInseridos;
54
55          if (buffer.capacity() < tamanho){
56              System.out.println("Erro: Estouro do buffer");
57          }
58
59          capacidadeInicial = (byte) buffer.remaining();
60          buffer = buffer.putFloat(tamanho, dado);
61          capacidadeFinal = (byte) buffer.remaining();
62
63          bytesInseridos = (byte)(capacidadeFinal -
64                                          capacidadeInicial);
65
66          m_ixwr = buffer.limit();
67          m_ixrd = buffer.position();
68
69        return bytesInseridos;
70      }
71
72      @Override
73      public byte remover(float dado, byte tamanho) {
74
75          if (m_ixwr == 0) {
76                  throw new NoSuchElementException();
77             }
78
79          byte capacidadeInicial;
80          byte capacidadeFinal;
81          byte bytesRemovidos;
82          float dadoRemovido;
83
84          capacidadeInicial = (byte) buffer.remaining();
85          dadoRemovido = buffer.getFloat();
86          capacidadeFinal = (byte) buffer.remaining();
87          bytesRemovidos = (byte)(capacidadeFinal -
88                                          capacidadeInicial);
89
90          m_ixwr = buffer.limit();
91          m_ixrd = buffer.position();
92
93        return bytesRemovidos;
94      }
95
96
97
98      @Override
99      public byte tamanho() {
100
101         byte capacidade = (byte) buffer.capacity();
102         byte livre = (byte) buffer.remaining();
```

```
103
104             m_tamanho = (byte)(capacidade - livre);
105
106
107             return m_tamanho;
108         }
109
110         private int limit( ByteBuffer buffer) {
111
112            return buffer.limit();
113         }
114
115
116
117
118
119 }
```

- `GerenciadorDados.java`

```
1  package dados;
2  import java.util.GregorianCalendar;
3  import suporte.Comunicador;
4
5  /**
6   *
7   * @author Camila Achutti
8   */
9
10 //Coordena a manipulacao de dados dos buffers: cientifico,
11 // diagnostico, housekeeping, carga de dados, descarga de
12 // dados e relato de eventos. singleton
13 public class GerenciadorDados implements IGerenciadorDados {
14
15         private BufferSimples m_bufCco; //Buffer simples para
16                                 // armazenamento de temperaturas.
17         private BufferSimples m_bufEvt;
18         //Buffer simples (memoria nao paginada) para relato
19         //de eventos.
20         private static GerenciadorDados instancia = null;
21
22       private GerenciadorDados (){
23
24       }
25
26       public static GerenciadorDados instanciar(){
27          if (instancia == null) {
28              instancia = new GerenciadorDados();
29          }
30        return instancia;
31      }
```

```
32
33      @Override
34      public void iniciar() {
35          byte capacidade= (byte)1024;
36          m_bufEvt = new BufferSimples( this , capacidade);
37
38      }
39
40      @Override
41      public double adicionar(TipoBufferEnum destino,
42                                  int dado, byte tamanho) {
43          byte tamanho_evento_adicionado;
44          switch (destino) {
45              case tbCientifico:
46                  break;
47              case tbDiagnose:
48                  break;
49              case tbTeste:
50                  break;
51              case tbHousekeeping: case tbTemperatura:
52                  break;
53              case tbRelatoEventos:
54                 tamanho_evento_adicionado =
55                         m_bufEvt.inserir(dado,tamanho);
56                 return tamanho_evento_adicionado;
57              default:
58                  break;
59          }
60
61          return 0;
62      }
63
64      @Override
65      public void relatarEvento(TipoRelatoEventoEnum tipo,
66                                              int[] info){
67          RelatoEvento relatoEvento = new RelatoEvento();
68          relatoEvento.idTipo = tipo;
69          relatoEvento.info = info;
70
71
72      }
73
74  public int getHora() {
75
76  // cria um StringBuilder
77  StringBuilder sb = new StringBuilder();
78
79  // cria um GregorianCalendar que vai conter a hora atual
80  GregorianCalendar d = new GregorianCalendar();
81
82  // anexa do StringBuilder os dados da hora
```

```
83  sb.append( d.get( GregorianCalendar.HOUR_OF_DAY ) );
84  sb.append( d.get( GregorianCalendar.MINUTE ) );
85  sb.append( d.get( GregorianCalendar.SECOND ) );
86
87  String hora = sb.toString();
88  int horaInt = Integer.parseInt(hora);
89
90   return horaInt;
91      }
92
93  @Override
94      public double remover(TipoBufferEnum local, int dado,
95                                              byte tamanho) {
96          byte tamanho_evento_removido;
97           switch (local) {
98               case tbCientifico:
99                   break;
100              case tbDiagnose:
101                  break;
102              case tbTeste:
103                  break;
104              case tbHousekeeping: case tbTemperatura:
105                  break;
106              case tbRelatoEventos:
107                 tamanho_evento_removido = m_bufEvt.
108                                 inserir(dado, tamanho);
109                 return tamanho_evento_removido;
110              default:
111                  break;
112          }
113          return 0;
114      }
115 }
```

- `Housekeeper.java`

```
1  package dados;
2
3  import java.io.File;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.io.PrintWriter;
7  import java.text.DateFormat;
8  import java.text.SimpleDateFormat;
9  import java.util.Date;
10 import java.util.logging.Level;
11 import java.util.logging.Logger;
12 import suporte.Iniciador;
13 /**
14  *
15  * @author Camila Achutti
```

```
16  */
17
18  //Tarefa responsavel pela geracao e formatacao de dados
19  // de housekeeping.
20  public class Housekeeper {
21
22      //Memoria de trabalho para composicao do pacote de
23      // housekeeping.
24      private PacoteHK m_pacHke = new PacoteHK();
25      String caminhoAbsolutoRelatorio =
26          "/Users/cachutti/Desktop/IC/teste/relatorio.txt";
27
28      //Ponto de entrada da tarefa.
29      public void executar() throws IOException{
30
31          prepararPacoteHK();
32          formatarRelatorio();
33          }
34
35
36      //Executa a preparacao do pacote de housekeeping,
37      //acessando as origens de dados para um housekeeping
38      // completo, gravando os dados no pacote de
39      //housekeeping interno (m_pacHke).
40      private void prepararPacoteHK(){
41
42          m_pacHke.amoTta [0] = 0;
43          m_pacHke.amoTta [1] = 1;
44          m_pacHke.amoTta [2] = 2;
45          m_pacHke.amoTta [3] = 3;
46          m_pacHke.amoTta [4] = 4;
47          m_pacHke.amoTta [5] = 5;
48          m_pacHke.amoTta [6] = 6;
49          m_pacHke.amoTta [7] = 7;
50          m_pacHke.amoTta [8] = 8;
51          m_pacHke.amoTta [9] = 9;
52          m_pacHke.amoTta [10] = 10;
53          m_pacHke.amoTta [11] = 11;//Ultimas 12 amostras
54                                    //de temperaturas
55          m_pacHke.modOpe = "Nominal"; //Modo de operacao
56                                       // corrente do SWPDC.
57          m_pacHke.tpoAmoTta = 0; //Tempo de amostragem de
58                                          // temperaturas.
59          m_pacHke.qteErrSim = 0; //Quantidade de erros
60             //simples ocorridos desde o ultimo zeramento.
61          m_pacHke.relEvt = TipoRelatoEventoEnum.treBufDdoCco10;
62          //Ultimos relatos de eventos ocorridos.
63          }
64
65      private void formatarRelatorio () throws IOException{
66      try{
```

```
67          File arquivo = new File( caminhoAbsolutoRelatorio );
68          arquivo.createNewFile();
69          FileWriter arquivoW = new FileWriter (arquivo);
70          PrintWriter printW = new PrintWriter (arquivoW);
71          printW.println("Relatorio PacoteHK");
72          printW.println();
73          printW.println("[data] " + getData() + "       [hora] "
74                                                  + getTime());
75          printW.println("[modoOpSWPDC] " + m_pacHke.modOpe);
76          printW.println();
77          printW.println("Amostras de temperaturas");
78          printW.println(m_pacHke.amoTta[0]);
79          printW.println(m_pacHke.amoTta[1]);
80          printW.println(m_pacHke.amoTta[2]);
81          printW.println(m_pacHke.amoTta[3]);
82          printW.println(m_pacHke.amoTta[4]);
83          printW.println(m_pacHke.amoTta[5]);
84          printW.println(m_pacHke.amoTta[6]);
85          printW.println(m_pacHke.amoTta[7]);
86          printW.println(m_pacHke.amoTta[8]);
87          printW.println(m_pacHke.amoTta[9]);
88          printW.println(m_pacHke.amoTta[10]);
89          printW.println(m_pacHke.amoTta[11]);
90          printW.println(m_pacHke.qteErrSim + " Erros");
91          printW.println("Tempo de amostragem: " +
92                                      m_pacHke.tpoAmoTta );
93          printW.println();
94          printW.println("Historico eventos");
95          printW.println(m_pacHke.relEvt);
96          printW.flush();
97         printW.close();
98      } catch (IOException e) {
99          Logger.getLogger(Iniciador.class.getName()).
100                                 log(Level.SEVERE, null, e);
101     }
102 }
103
104     private String getData() {
105         DateFormat dateFormat =
106                         new SimpleDateFormat("dd/MM/yyyy");
107         Date date = new Date();
108         return dateFormat.format(date);
109     }
110
111     private String getTime() {
112         DateFormat dateFormat =
113                             new SimpleDateFormat("HH:mm:ss");
114         Date date = new Date();
115         return dateFormat.format(date);
116     }
117 }
```

Capítulo 4

Validação do sistema

4.1 Método

A validação do sistema SWPDC desenvolvido foi realizada baseada em cenários de execução reais produzidos no contexto do projeto de Qualidade de Software Embarcado em Aplicações Espaciais (QSEE) do Instituto Nacional de Pesquisas Espaciais (INPE). Contamos com responsáveis técnicos que garantiram a qualidade e a relevância dos cenários produzidos.

Atuamos nesse trabalho sobre dois cenários - 17 e 50 - representados abaixo pelos seus Statescharts desenvolvidos pelo Doutor Valdivino Santiago Júnior durante o seu trabalho de doutorado [39]:

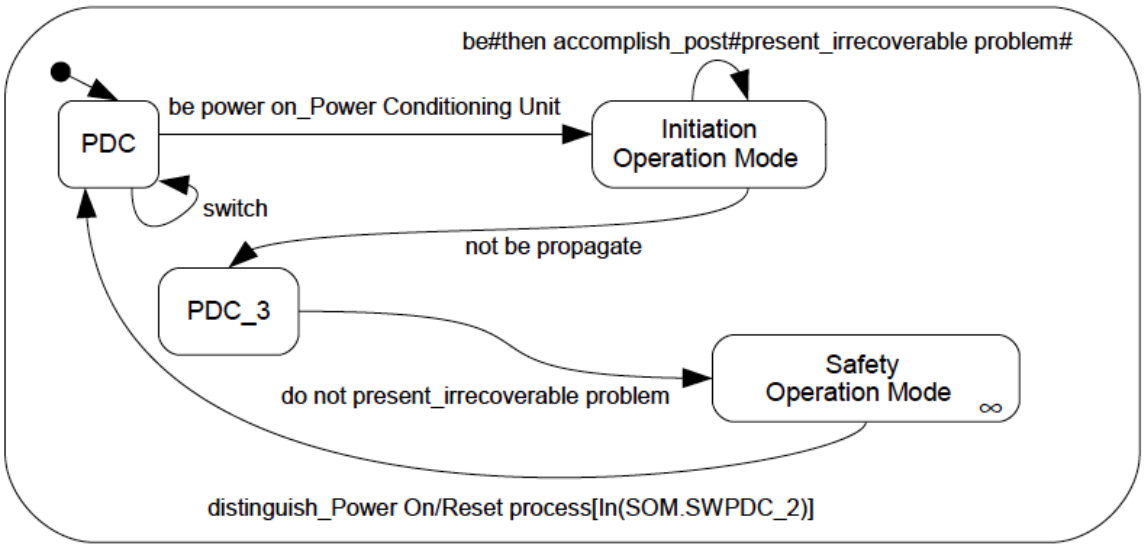

Figura 4.1: Statechart do comportamento normal do cenário 50

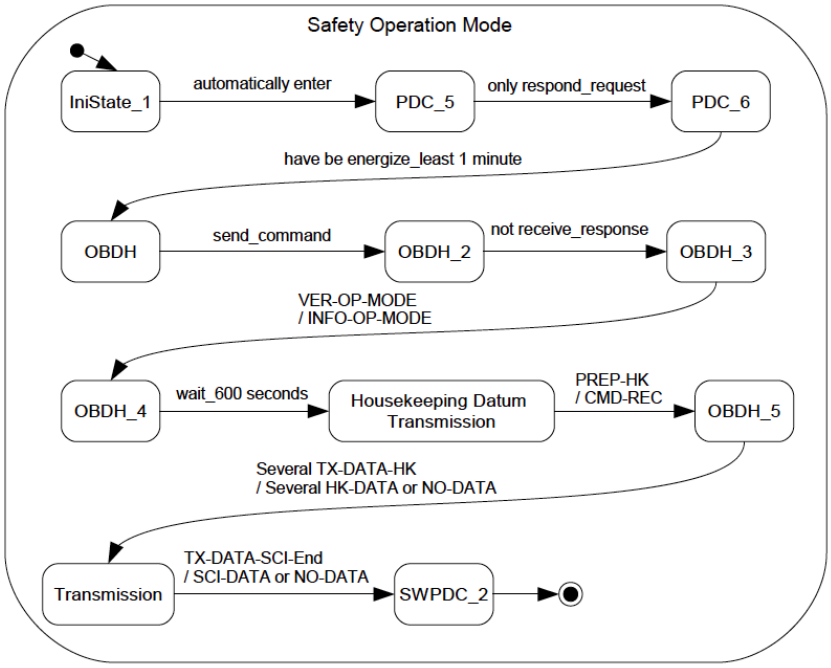

Figura 4.2: Safety Operation Mode - cenário 50

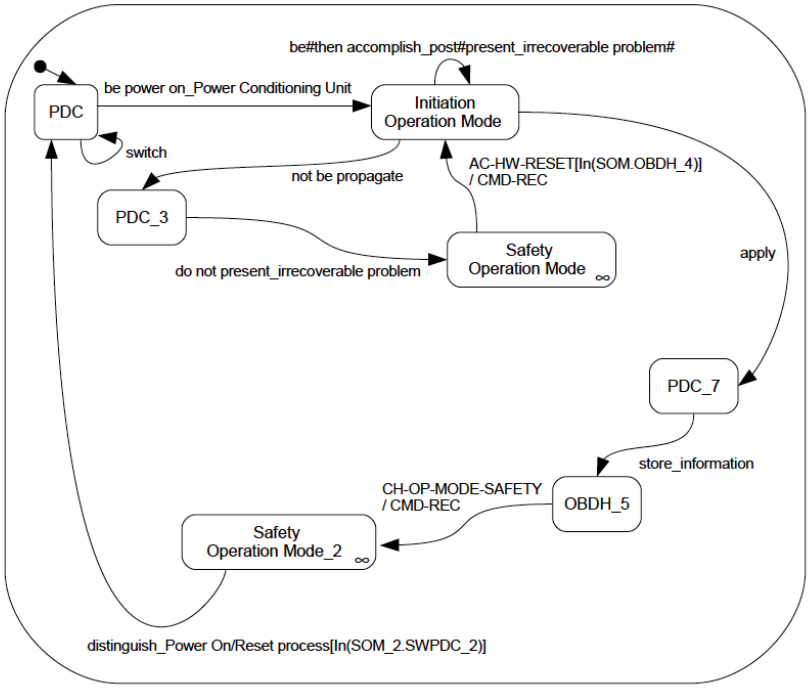

Figura 4.3: Statechart do comportamento normal do cenário 17

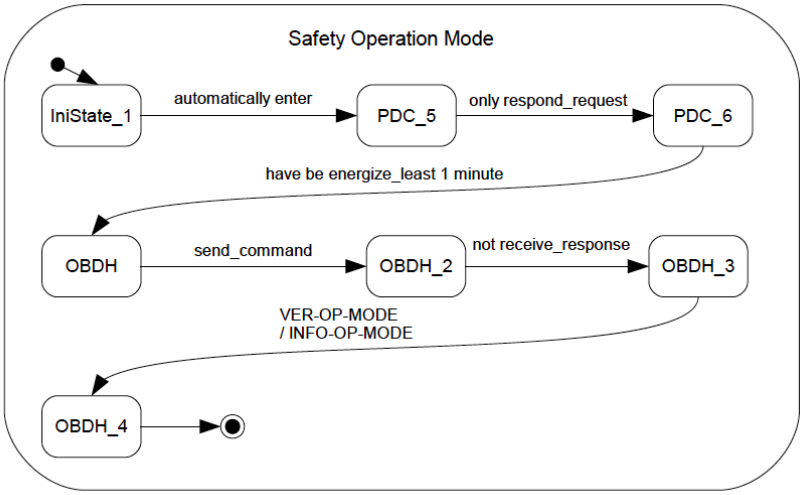

Figura 4.4: Safety Operation Mode 1- cenário 17

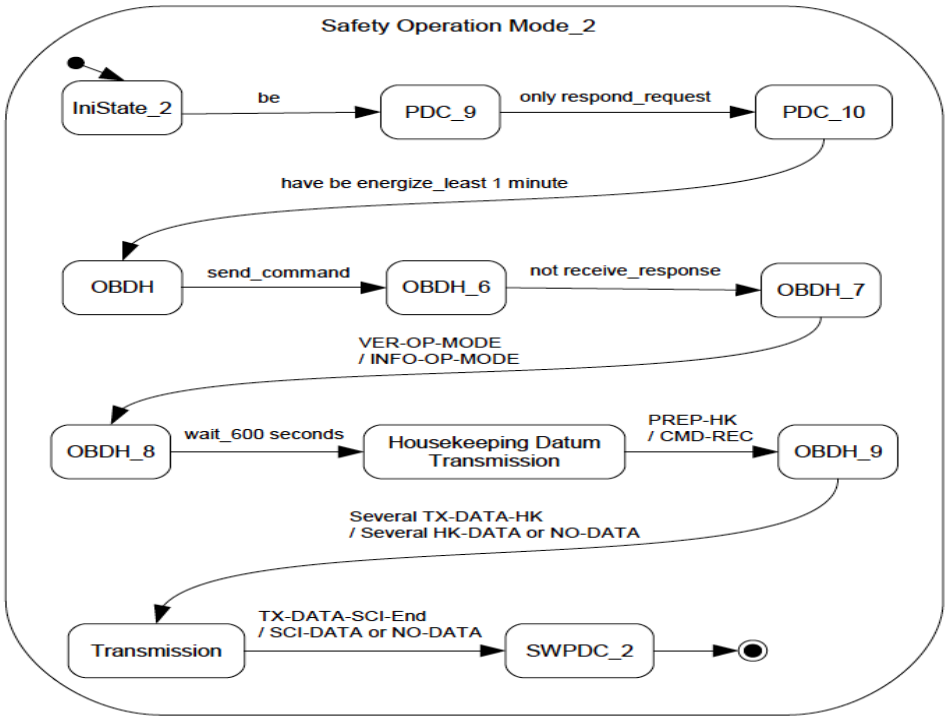

Figura 4.5: Safety Operation Mode 2- cenário 17

Os requisitos representados nesses statecharts estão mapeados nas Tabelas 4.1 e 4.2.

Tabela 4.1: Tradução do cenários

Código	Descrição Inglês	Tradução
SRS001	The PDC shall be powered on by the Power Conditioning Unit	O PDC será ligado pela Unidade de condicionamento de energia
SRS002	The PDC shall be in the Initiation Operation Mode after being powered on. The SWPDC shall then accomplish a POST. If PDC presents any irrecoverable problem, this computer shall remain in the Initiation Operation Mode and such a problem shall not be propagated to the OBDH.	O PDC entrará no modo de operação Iniciação depois de ser ligado. O SWPDC deve então realizar um POST. Se o PDC apresenta qualquer problema irrecuperável , o computador permanecerá no modo de operação de Iniciação e tal problema não deve ser propagado para o OBDH
SRS003	If PDC does not present any irrecoverable problem, after the initiation process, the PDC shall automatically enter into the Safety Operation Mode.	Se o PDC não apresentar qualquer problema irrecuperável , após o processo de iniciação, o PDC entra automaticamente no modo de operação de NOMINAL
POCP001	The PDC can only respond to requests (commands) from OBDH after the PDC has been energized for at least 1 minute. If OBDH sends commands within less than 1 minute, the OBDH shall not receive any response from PDC.	O PDC só pode responder às solicitações (comandos) de OBDH após o PDC foi energizado por pelo menos 1 minuto. Se OBDH envia comandos em menos de 1 minuto, o OBDH não receber qualquer resposta do PDC
RB001	The OBDH shall send VEROPMODE to PDC.	O OBDH enviará VEROPMODE para PDC.
SRS004	The OBDH should wait 600 seconds before asking for a Housekeeping Data frame.	O OBDH deve esperar 600 segundo antes de pedir um quadro os dados de housekeeping
SRS005	Housekeeping data transmission shall start with PREPHK. After that, the OBDH can send several TX-DATAHK to PDC. The transmission shall be ended with TXDATASCIEnd.	Housekeeping data transmission shall start with PREP - HK. Depois que , a OBDH pode enviar vários TX-DATA-HK para PDC. A transmissão deve ser encerrada com TX-DATA-SCI-End.
SRS009	The SWPDC shall distinguish between Power On/Reset processes.	O SWPDC devem distinguir entre processos de Power On / Reset
RB005	After switching both EPPHxs off via PDC, the OBDH shall switch the PDC off via the Power Conditioning Unit	Depois de mudar tanto EPPHxs off via PDC , o OBDH deve mudar o PDC off através da unidade de condicionamento de energia

Tabela 4.2: Adaptação dos cenários para produção dos casos de teste no nosso contexto

Código	Adaptação	Casos de teste
SRS001	ADAPTADA - O SWPDC será startado e verificará a situação do simulador. Este já deve ter sido inicializado e se encontrar no estado NOMINAL para que a troca de comando e dados seja iniciada.	[TESTE] SRS009-1, SRS009-2, SRS009-3, RB001-1, RB001-2, RB001-3, RB001-4
SRS002	ADAPTADA - O SWPDC só começa depois de dar um POST no comando para verificar o estado do PDC (INFORMAR MODO DE OPERAÇÃO) e só vai começar a "usá-lo"se a resposta for a de que está tudo corretamente inicializado - ANÁLOGO AO REQUISITO SRS001 ADAPTADO	[TESTE] SRS009-1, SRS009-2, SRS009-3, RB001-1, RB001-2, RB001-3, RB001-4
SRS003	MANTIDA - Se o PDC não apresentar qualquer problema irrecuperável , após o processo de iniciação, o PDC entra automaticamente no modo de operação de NOMINAL	[TESTE] RB001-1, RB001-2, RB001-3, RB001-4
POCP01	ADAPTADA - O PDC só pode responder às solicitações (comandos) de SWPDC após o PDC ter sido energizado por pelo menos 1 minuto. Se o SWPDC envia comandos em menos de 1 minuto, ele não recebe qualquer resposta do PDC.	[SIMULAÇÃO] POCP01-1, POCP01-2, POCP01-3
RB001	ADAPTADA - O SWPDC enviará INFORMAR ESTADO para PDC	[TESTE] RB001-1, RB001-2, RB001-3, RB001-4
SRS004	MANTIDA - O SWPDC deve esperar pelo menos 600 segundo antes de pedir dados de housekeeping para o PDC	[SIMULAÇÃO] SRS004-1, SRS004-2, SRS004-3
SRS005	ADAPTADA - A transmissão de 12 amostras deve ser realizada a cada requisição.	[TESTE] SRS005-1, SRS005-2, SRS005-3, SRS005-4, SRS005-5
SRS009	MANTIDA - O SWPDC devem distinguir entre processos de Power On / Reset	[TESTE] SRS009-1, SRS009-2, SRS009-3
RB005	Não se aplica	-

Eles foram traduzidos do inglês para o português e adaptados para o nosso ambiente e escopo de desenvolvimento e simulação. Com base nestes construímos casos de teste mais elaborados que cobrissem os requisitos específicados. Dois dos requisitos foram validados usando o simulador do PDC produzido.

Com base nesse mapeamento e se valendo de recursos como teste unitário automatizado e *Mock Objects* (objetos que simulam o comportamento de objetos reais de forma controlada) validamos cada um deles. Para a escrita e execução dos citados testes fizemos uso de 3 ferramentas: **JUnit**, framework para desenvolvimento de teste automatizado já abordado previamente nesse trabalho; **Eclemma**, uma ferramenta de análise de cobertura de código, também já abordade nesse trabalho; **Mockito**, biblioteca para criação de Mock Objects, tem sintaxe agradável e por isso foi escolhida entre as várias outras opções que temos de bibliotecas de mock em java.

4.2 Casos de teste para o requisito SRS005

Os casos de teste desenvolvidos para o requisito SRS005 foram: SRS005-1 (Tabela 4.3), SRS005-2 (Tabela 4.4), SRS005-3 (Tabela 4.5), SRS005-4 (Tabela 4.6), SRS005-5 (Tabela 4.7)

Tabela 4.3: Caso de teste SRS005-1

ID	**SRS005-1**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testObterAmostraSucesso()
Funcionalidade	AQUISIÇÃO BEM SUCEDIDA
Pré-condição	SWPDC em NOMINAL; Emissão de comando de aquisição; PDC em estado consistênte
Procedimento	Limpeza do buffer; Adquirir temperaturas; Verificar se cada amostra pertence ao intervalo; Armazenar as corretas no buffer
Resultado Esperado	Armazenamento das amostrar corretas no buffer e guardar comportamento no histórico

Tabela 4.4: Caso de teste SRS005-2

ID	**SRS005-2**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testObterAmostraNaoAtual()
Funcionalidade	AQUISIÇÃO MAL SUCEDIDA POR ATRASO DO PDC
Pré-condição	SWPDC em NOMINAL; Emissão de comando de aquisição; PDC em estado inconsistente (ATRASO de amostra)
Procedimento	Limpeza do buffer; Adquirir temperaturas FALHA;
Resultado Esperado	Abortar armazenamento, log do erro e guardar no histório comportamento

Tabela 4.5: Caso de teste SRS005-3

ID	**SRS005-3**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testObterAmostraNull()
Funcionalidade	AQUISIÇÃO MAL SUCEDIDA AMOSTRAGEM VAZIA
Pré-condição	SWPDC em NOMINAL; Emissão de comando de aquisição; PDC em estado inconsistente (amostra vazia)
Procedimento	Limpeza do buffer; Adquirir temperaturas FALHA;
Resultado Esperado	Abortar armazenamento, log do erro e guardar no histório comportamento

Tabela 4.6: Caso de teste SRS005-4

ID	**SRS005-4**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testObterAmostraExcecao()
Funcionalidade	AQUISIÇÃO MAL SUCEDIDA POR FALHA NO ARQUIVO DE AMOSTRAS
Pré-condição	SWPDC em NOMINAL; Emissão de comando de aquisição; PDC em estado consistente; Escrita no arquivo falhou
Procedimento	Limpeza do buffer; Adquirir temperaturas FALHA;
Resultado Esperado	Abortar armazenamento, log do erro e guardar no histório comportamento

Tabela 4.7: Caso de teste SRS005-5

ID	**SRS005-5**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testObterAmostraNaoNumAm()
Funcionalidade	AQUISIÇÃO MAL SUCEDIDA NÚMERO ERRADO DE AMOSTRAS
Pré-condição	SWPDC em NOMINAL; Emissão de comando de aquisição; PDC em estado inconsistente (número errado de amostras)
Procedimento	Limpeza do buffer; Adquirir temperaturas FALHA;
Resultado Esperado	Abortar armazenamento, log do erro e guardar no histório comportamento

4.3 Casos de teste desenvolvidos a partir do requisito SRS009

Os casos de teste desenvolvidos pra o requisito SRS009 foram: SRS009-1 (Tabela 4.8), SRS009-2 (Tabela 4.9), SRS009-3 (Tabela 4.10).

Tabela 4.8: Caso de teste SRS009-1

ID	**SRS009-1**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testVerificaStatusAlimentacaoLigadoSucesso()
Funcionalidade	RECONHECIMENTO DE STATUS DE ALIMENTAÇÃO - (Ligado, Desligado)
Pré-condição	inicialização SWPDC bem sucedida, arquivo de sysInfo consistente
Procedimento	SWPDC emite comando de verificação de status; Emissão bem sucedida; Recupera status; Diferenciação do status; Status ligado reconhecido.
Resultado Esperado	Reconhecimento do status ligado, progresso na recuperação de estado do PDC e guardar comportamento no histórico

Tabela 4.9: Caso de teste SRS009-2

ID	**SRS009-2**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testVerificaStatusAlimentacaoDiferenteDesligado()
Funcionalidade	RECONHECIMENTO DE STATUS DE ALIMENTAÇÃO - (Ligado, Desligado, Reset)
Pré-condição	inicialização SWPDC bem sucedida, arquivo de sysInfo consistente
Procedimento	SWPDC emite comando de verificação de status; Emissão bem sucedida; Recupera status; Diferenciação do status; Status desligado reconhecido.
Resultado Esperado	Reconhecimento do status desligado, cancelamento da recuperação de estado do PDC e guardar comportamento no histórico

Tabela 4.10: Caso de teste SRS009-3

ID	**SRS009-3**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: testVerificaStatusAlimentacaoDiferenteExcecao()
Funcionalidade	RECONHECIMENTO DE STATUS DE ALIMENTAÇÃO - (Ligado, Desligado, Rest)
Pré-condição	inicialização SWPDC bem sucedida, arquivo de sysInfo consistente
Procedimento	SWPDC emite comando de verificação de status; Emissão bem sucedida; Recupera status; Diferenciação do status; Status reset reconhecido.
Resultado Esperado	Reconhecimento do status ligado, pausa na recuperação de estado do PDC e guardar comportamento no histórico

4.4 Casos de teste para o requisito RB001

Os casos de teste desenvolvidos para o requisito RB001 foram: RB001-1 (Tabela 4.11), RB001-2 (Tabela 4.12), RB001-3 (Tabela 4.13), RB001-4 (Tabela 4.14).

Tabela 4.11: Caso de teste RB001-1

ID	**RB001-1**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: test-VerificaEstadoQuandoLeModoOp()
Funcionalidade	ACOMPANHAMENTO DO ESTADO DO PDC (Inicialização, Nominal)
Pré-condição	inicialização SWPDC bem sucedida, inicialização PDC bem sucedida, arquivo de sysInfo consistente,
Procedimento	SWPDC emite comando de verificação de estado; Emissão bem sucedida; Recupera estado; Diferenciação do estado; Status ligado reconhecido.
Resultado Esperado	Reconhecimento do status Nominal, progresso para as atividades de gerenciamento de dados e guardar comportamento no histórico

Tabela 4.12: Caso de teste RB001-2

ID	**RB001-2**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: test-VerificaEstadoQuandoLeNull()
Funcionalidade	ACOMPANHAMENTO DO ESTADO DO PDC (Inicialização, Nominal)
Pré-condição	inicialização SWPDC bem sucedida, inicialização PDC bem sucedida, arquivo de sysInfo inconsistente,
Procedimento	SWPDC emite comando de verificação de estado; Emissão bem sucedida; Recupera estado; Estado NULL;
Resultado Esperado	Reconhecer erro, cancelar atividades de gerenciamento de dados e guardar comportamento no histórico.

Tabela 4.13: Caso de teste RB001-3

ID	**RB001-3**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: test-VerificaEstadoQuandoNaoLeModoOp()
Funcionalidade	ACOMPANHAMENTO DO ESTADO DO PDC (Inicialização, Nominal)
Pré-condição	inicialização SWPDC bem sucedida, inicialização PDC bem sucedida, arquivo de sysInfo inconsistente,
Procedimento	SWPDC emite comando de verificação de estado; Emissão mal sucedida.
Resultado Esperado	Reconhecer erro, cancelar atividades de gerenciamento de dados e guardar comportamento no histórico.

Tabela 4.14: Caso de teste RB001-4

ID	**RB001-4**
Teste	classe: SWPDC.test.suporte.VerificadorTest teste: test-VerificaEstadoExcecao()
Funcionalidade	ACOMPANHAMENTO DO ESTADO DO PDC (Inicialização, Nominal)
Pré-condição	inicialização SWPDC bem sucedida, inicialização PDC bem sucedida, arquivo de sysInfo inconsistente,
Procedimento	Emite comando de verificação de estado; Problema com a leitura ou escrita nos arquivos
Resultado Esperado	Reconhecer erro, cancelar atividades de gerenciamento de dados e guardar comportamento no histórico.

4.5 Simulação dos requisitos POCP01 e SRS004

O requisito POPC01 quer garantir que o SWPDC somente começará a interagir com o PDC depois de transcorridos pelo menos 60 segundos desde a inicialização do PDC. Já o requisito SRS004 quer garantir que o SWPDC vai requisitar dados de housekeeping de 10 em 10 minutos para o PDC. Para validarmos esses requistos desenvolvemos os seguintes casos para a simulação sumarizados nas Tabelas 4.15, 4.16:

Tabela 4.15: Simulações requisito POPC01

ID Caso	**Parâmetros de simulação**
POPC01-1	tempo até a primeira interação entre SWPDC e PDC < 60s
POPC01-2	tempo até a primeira interação entre SWPDC e PDC $= 60$s
POPC01-3	tempo até a primeira interação entre SWPDC e PDC $>> 60$s

Tabela 4.16: Simulações requisito SRS004

ID Caso	**Parâmetros de simulação**
SRS004-2	intervalo de requisição de dados de housekeeping do SWPDC para o PDC = 10min
SRS004-3	intervalo de requisição de dados de housekeeping do SWPDC para o PDC > 10min

Para realizar as simulações descritas acima tivemos que alterar o código fonte do SWPDC manualmente e observar seu comportamento. Tal procedimento foi adotado, pois eram poucos os casos de simulação e temos acesso ao código fonte e ao seu ambiente de desenvolvimento, tornando essa possibilidade não só viável como também a mais rápida de ser realizada.

A avaliação do comportamento do SWPDC em cada uma das situações foi realizado comparando os relatórios do Pacote HK gerados durante elas e o relatório que deveria ter

sido gerado para as amostra aleátorias geradas pelo PDC. Para cada um dos casos, tanto o simulador com o SWPDC era reinicializado, afinal para cada proceder com cada um deles precisamos alterar o código e realizar o build novamente.

Todos os casos foram bem sucedidos, então para nos certificarmos de que o SWPDC tinha de fato se comportado bem durante a simulação de ambos os requisitos, acompanhamos também o arquivo de histórico de execução. A principal observação realizada aqui deve ser os intervalos de tempo e a ausência de qualquer registro de erro, exatamente como esperado para validação de ambos os requisitos. Abaixo estão os arquivos obtidos na simulação:

- Histórico obtido na simulação POPC01-1, onde a primeira requisição ocorre juntamento com a inicialização do PDC. O resultado foi consistente uma vez que o SWPDC aguardou corretamente a inicialização do PDC refazendo a requisição de interação a cada 20 minutos:

```
05/11/2013 07:54:51.105 05/11/2013 07:54:51.104 emitiu comando:
COMEÇAR SIMULAÇAO
05/11/2013 07:54:51.105 PDC inicializado com sucesso.
05/11/2013 07:54:51.106 05/11/2013 07:54:51.106 emitiu comando:
INFORMAR ESTADO
05/11/2013 07:54:51.107 05/11/2013 07:54:51.106 Aguardar PDC.
05/11/2013 07:54:81.106 05/11/2013 07:54:51.106 emitiu comando:
INFORMAR ESTADO
05/11/2013 07:54:81.107 05/11/2013 07:54:51.106 Aguardar PDC.
05/11/2013 07:55:11.106 05/11/2013 07:54:51.106 emitiu comando:
INFORMAR ESTADO
05/11/2013 07:55:11.107 05/11/2013 07:54:51.106 Aguardar PDC.
05/11/2013 07:55:41.106 05/11/2013 07:54:51.106 emitiu comando:
INFORMAR ESTADO
05/11/2013 07:55:41.107 05/11/2013 07:54:51.106 Aguardar PDC.
05/11/2013 07:55:51.106 Verificação de estado do PDC realizada com
sucesso - status: nominal
05/11/2013 07:55:51.107 PDC está no estado NOMINAL. Inicialização
bem sucedida
05/11/2013 07:55:51.618 05/11/2013 07:54:51.618 emitiu comando:
OBTER TEMPERATURA
05/11/2013 07:55:56.619 Obtenção das novas amostras de temperatura
05/11/2013 07:55:56.620 Amostras obtidas com sucesso
05/11/2013 07:55:56.624 Foram detectados 4 dados errados durante o
armazenamento da amostra 1
```

- Histórico obtido na simulação POPC01-2, onde a requisição acontece após transcorrido exatamento 1 minutos. O resultado foi consistente e essa é a situação de trabalho padrão do conjunto SWPDC+PDC:

```
05/11/2013 07:46:14.347 05/11/2013 07:46:14.347 emitiu comando:
COMEÇAR SIMULAÇAO
05/11/2013 07:46:14.347 PDC inicializado com sucesso.
05/11/2013 07:47:14.349 05/11/2013 07:47:14.348 emitiu comando:
INFORMAR ESTADO
05/11/2013 07:47:14.349 Verificação de estado do PDC realizada com
sucesso - status: nominal
05/11/2013 07:47:14.350 PDC está no estado NOMINAL. Inicialização
bem sucedida
05/11/2013 07:47:14.857 05/11/2013 07:46:14.857 emitiu comando:
OBTER TEMPERATURA
05/11/2013 07:47:19.858 Obtenção das novas amostras de temperatura
05/11/2013 07:47:19.859 Amostras obtidas com sucesso
05/11/2013 07:47:19.861 Foram detectados 4 dados errados durante o
armazenamento da amostra 1
```

- Histórico obtido na simulação POPC01-3, onde a requisição acontece somente após transcorridos 20 minutos da inicialização do PDC. O resultado resultado abtido também foi consistente como pode ser observado abaixo:

```
05/11/2013 08:46:14.347 05/11/2013 08:46:14.347 emitiu comando:
COMEÇAR SIMULAÇAO
05/11/2013 08:46:14.347 PDC inicializado com sucesso.
05/11/2013 09:06:14.347 05/11/2013 09:06:14.348 emitiu comando:
INFORMAR ESTADO
05/11/2013 09:06:14.349 Verificação de estado do PDC realizada com
sucesso - status: nominal
05/11/2013 09:06:14.350 PDC está no estado NOMINAL. Inicialização
bem sucedida
05/11/2013 09:06:14.857 05/11/2013 07:46:14.857 emitiu comando:
OBTER TEMPERATURA
05/11/2013 09:06:19.858 Obtenção das novas amostras de temperatura
05/11/2013 09:06:19.859 Amostras obtidas com sucesso
05/11/2013 09:06:19.861 Foram detectados 4 dados errados durante o
armazenamento da amostra 1
```

- Histórico obtido na simulação SRS004-1, onde o intervalo de requisição de dados de housekeeping foi de 10 minutos. O resultado foi consistente e essa é a situação de trabalho padrão do conjunto SWPDC+PDC. As requisições intermediárias por amostras foram suprimidas para melhor entendimento. Conforme pode ser observado abaixo:

```
05/11/2013 11:00:04.296 05/11/2013 11:00:04.295 emitiu comando:
COMEÇAR SIMULAÇAO
05/11/2013 11:00:04.297 PDC inicializado com sucesso.
05/11/2013 11:01:04.297 05/11/2013 11:01:04.297 emitiu comando:
INFORMAR ESTADO
05/11/2013 11:01:04.297 Verificação de estado do PDC realizada com
sucesso - status: nominal
05/11/2013 11:01:04.298 PDC está no estado NOMINAL. Inicialização
bem sucedida
05/11/2013 11:01:04.809 05/11/2013 11:00:04.809 emitiu comando:
OBTER TEMPERATURA
05/11/2013 11:01:09.811 Obtenção das novas amostras de temperatura
05/11/2013 11:01:09.813 Amostras obtidas com sucesso
05/11/2013 11:01:09.816 Foram detectados 4 dados errados durante o
armazenamento da amostra 1
.
.
.
05/11/2013 11:11:04.298 05/11/2013 11:11:04.297 emitiu comando:
INFORMAR ESTADO
05/11/2013 11:11:04.298 Verificação de estado do PDC realizada com
sucesso - status: nominal
05/11/2013 11:11:04.299 PDC está no estado NOMINAL. Inicialização
bem sucedida
```

- Histórico obtido na simulação SRS004-2, onde o intervalo de requisição de dados de housekeeping foi de 30 minutos. O resultado foi consistente, o que era de se esperar já que o PDC apenas responde comandos sem que qualquer verificação de intervalo seja feita, isso cabe ao escopo do SWPDC, que nessa simulação foi configurado para intervalar as requisições em 30 minutos conforme pode ser visto abaixo: (As requisições intermediárias por amostras foram suprimidas para melhor entendimento)

```
05/11/2013 11:32:05.912 05/11/2013 11:32:05.911 emitiu comando:
COMEÇAR SIMULAÇAO
05/11/2013 11:32:05.912 PDC inicializado com sucesso.
05/11/2013 11:32:05.912 05/11/2013 11:32:05.912 emitiu comando:
INFORMAR ESTADO
05/11/2013 11:32:05.913 Verificação de estado do PDC realizada com
sucesso - status: nominal
05/11/2013 11:32:05.913 PDC está no estado NOMINAL. Inicialização
bem sucedida
05/11/2013 11:32:06.421 05/11/2013 11:32:06.421 emitiu comando:
OBTER TEMPERATURA
```

```
05/11/2013 11:32:11.423 Obtenção das novas amostras de temperatura
05/11/2013 11:32:11.423 Amostras obtidas com sucesso
05/11/2013 11:32:11.427 Foram detectados 4 dados errados durante o
armazenamento da amostra 1
.
.
.
05/11/2013 12:02:05.914 05/11/2013 12:02:05.912 emitiu comando:
INFORMAR ESTADO
05/11/2013 12:02:05.915 Verificação de estado do PDC realizada com
 sucesso - status: nominal
05/11/2013 12:02:05.915 PDC está no estado NOMINAL. Inicialização
bem sucedida
```

Diante dos resultados apresentados o sistema SWPDC está validado e cumpre com os requisitos exigidos pelo contexto que está inserido. Obtemos assim, que o método de validação utilizado que se vale de cenários desenhados pelo cliente, se mostrou bastante confiável. Análise de ganho de cobertura de código será feito em seção subsequente.

Os testes não se resumem somente aos descritos aqui, que foram baseados nos cenários. Tivemos ainda a produção de testes baseados nas unidades produzidas, sendo também uma boa prática. Escrevemos testes que garantissem o correto funcionamento das funcionalidades das unidades e colaborassem com o processo de desenvolvimento do sistema.

4.6 Análise do método

Durante todo o desenvolvimento dos testes, monitoramos a cobertura de código do sistema SWPDC. Usamos essa médida como critério de avaliação da evolução da validação do sistema. Para realizar essa análise usamos a ferramenta Eclemma [**?**].

O Eclemma é uma ferramenta robusta que pode ser usada em programas Java de diferentes portes, este intrumenta diretamente o bytecode para realizar essa análise. Esta é dividida entre uma análise da cobertura geral, cobertura por pacotes e classes. Essa divisão na análise foi bastante valiosa para o nosso caso, onde temos algumas classes (interfaces e enumerations) que não são testadas por imposibilidades ou inutilidade de fazê-lo. Apesar disso afetar a cobertura geral, quando olhamos para a cobertura de pacotes e classes podemos realizar a análise dos resultados corretamente.

A título de comparação, realizamos alguns testes baseados somente no código produzido e nas principais funcionalidades do ponto de vista de desenvolvimento. Na avaliação do Eclemma obtivemos as seguintes parciais ilustradas na Figura 4.6 e resumidos na Tabela 4.17:

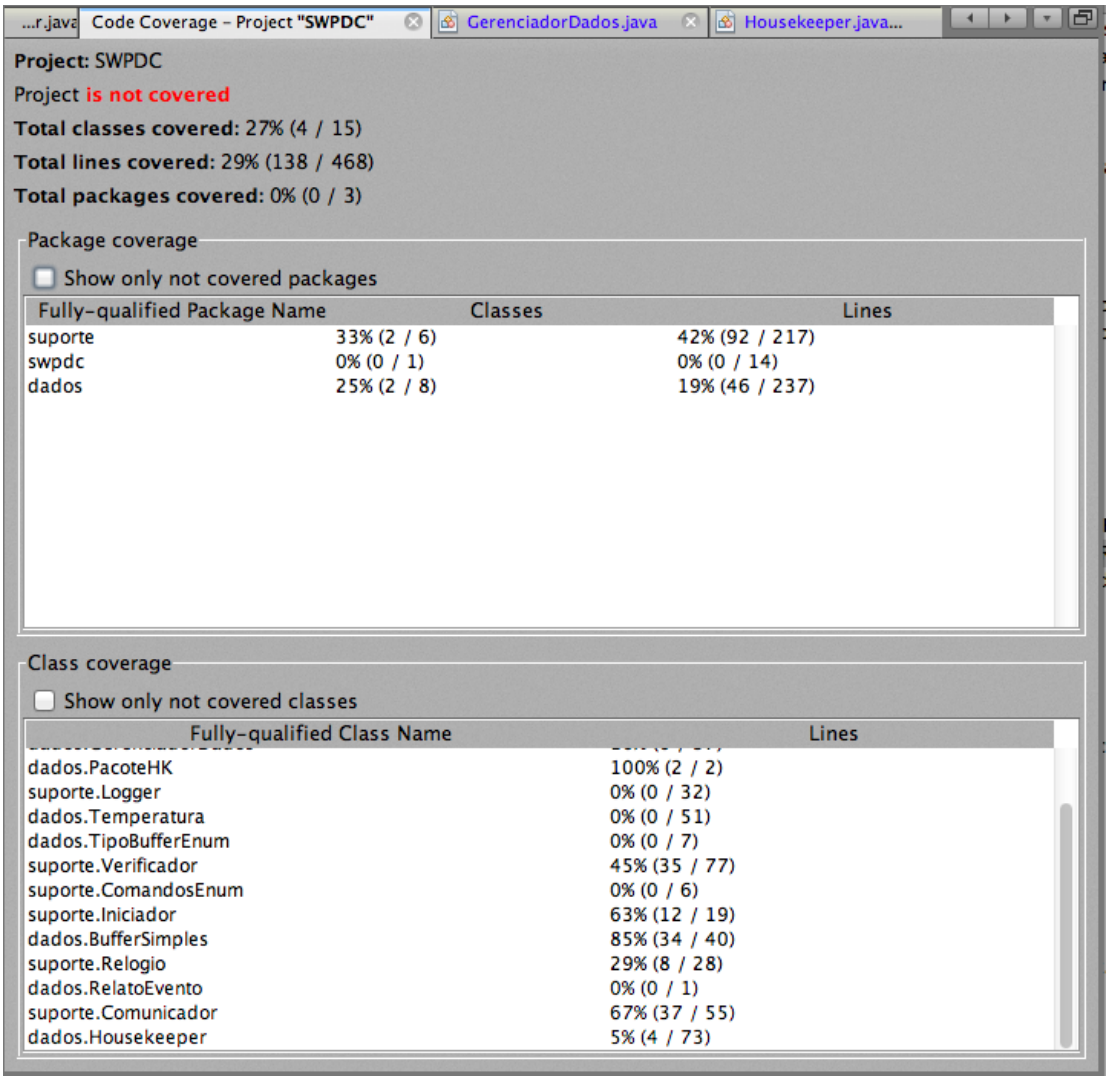

Figura 4.6: Modelagem do componente de Dados

Tabela 4.17: Resultados obtidos antes do estudo dos cenários.

Resultado	is NOT covered		
Classes cobertas	27%	**Cobertura do Pacote** `suporte`	33%
Linhas cobertas	29%	**Cobertura do Pacote** `swpdc`	0%
Pacotes cobertos	0%	**Cobertura do Pacote** `dados`	25%

A avaliação com Eclemma foi repetida ao fim da validação com cenários e o resultado final obtido está ilustrada na Figura 4.7 e resumido na Tabela 4.18:

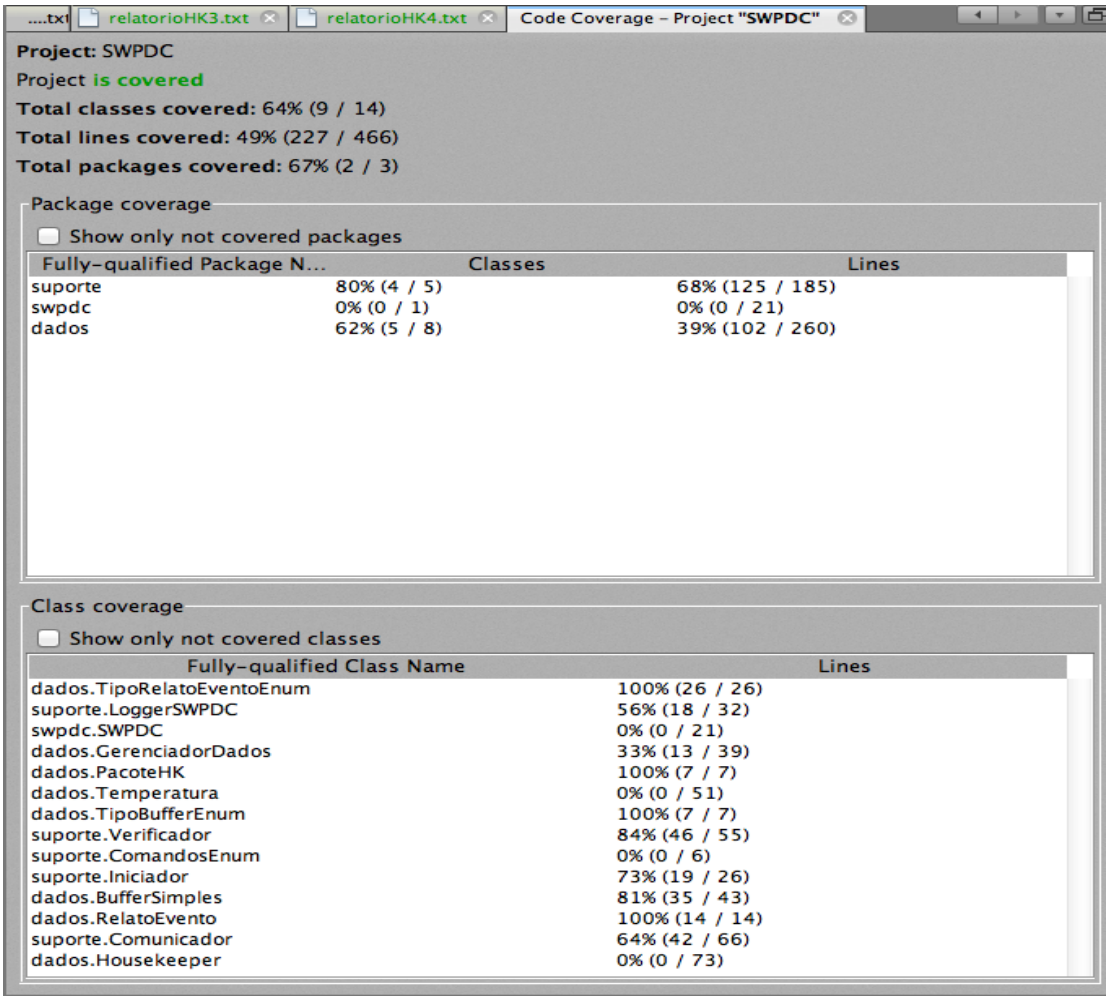

Figura 4.7: Modelagem do componente de Dados

Tabela 4.18: Resultados obtidos antes do estudo dos cenários.

Resultado	is covered		
Classes cobertas	64%	**Cobertura do Pacote** `suporte`	80%
Linhas cobertas	49%	**Cobertura do Pacote** `swpdc`	0%
Pacotes cobertos	67%	**Cobertura do Pacote** `dados`	62%

Olhando para as tabelas vemos ganhos significativos em todos os parâmetros de análise, mostrando a validade e os benefícios do uso do método proposto.

Cabe ressaltar que o pacote `swpdc` se manteve em 0% de cobertura por se tratar de um pacote com apenas uma classe que é um main requerido pelo JavaPathfinder (ferramenta

usada na verificação do programa), que somente chama as rotinas e não tem lógica própria, não sendo assim testável do ponto de vista prático.

Como conclusão da análise da validação realizada ressaltamos algumas dificuldades e benefícios da prática e do método utilizado:

- Dificuldades

  A principal dificuldade encontrada na validação desse projeto foi a necessidade de um cliente técnico com conhecimentos de ciência da computacão suficientes pra reconhecer requisitos essenciais do sistema e que pudesse transformá-los em cenários validáveis. Não é um requisito, ter um cliente técnico para realização desse tipo de validação, porém o processo é bastante facilitado com a presença de um.

  Outra dificuldade foi a realização da validação de requisitos que envolviam tempo e/ou intervalos de tempo. No trabalho desenvolvido, como foi desenvolvido um simulador de eventos tais requisitos foram validados por simulação, viabilizando tais validações.

- Benefícios

  Dentre os principais benefícios do método de validação desenvolvido nesse trabalho, temos o envolvimento do cliente no processo de validação sem a necessidade de interação cotidiana. Além disso, o cliente tem a necessidade de reconhecer quais são as necessidades do sistema e definir os escopos mais adequados para cada etapa do desenvolvimento.

  Além disso, com esse procedimento, validamos somente o que de fato faz sentido para o cliente, ganhando assim qualidade sem demanda de tempo desnecessária.

Capítulo 5

Verificação do sistema usando JPF

5.1 Método

A fim de realizar a verificação do nosso sistema usando o JavaPathfinder, algumas adequações precisaram ser realizadas. Estas foram depreendidas dos estudos e experimentos realizados durante testes com a ferramenta citada antes mesmo de começar o desenvolvimento do SWPDC, para que assim pudessemos já ter em mente algumas das características exigidas pelo JPF e assim diminuir o overhead para realizar a verificação num momento posterior a implementação. As principais características identificadas foram:

- O programa precisa ter um *main* bem definido, pois a partir da execução desse *main* ele vai percorrer os caminhos de execução que considera relevantes para fazer a verificação e evitar a explosão de estados;
- Evitar o uso das bibliotecas java.awt, java.net e java.io, pois estas não são suportadas pelo verificador. Para tais bibliotecas, o verificador até reconhece algumas funções, mas não todas.

Observando a segunda restrição, por ser algo que muitas vezes precisamos usar, a política adotada para lidar com ela foi: se existir a necessidade do uso de funções de alguma biblioteca que temos desconhecimento quanto a garantia de cobertura do *JPF*, será produzido um programa bem simples sobre o qual temos conhecimento pleno de seu funcionamento com tal função e realizaremos a verificação com o *JPF* para ver se ele acusa alguma incompatibilidade ou não cumpre a tarefa a que se propõe. Se não obtivermos qualquer problema nesse procedimento, faremos uso da função sem restrições. Entretanto, se existir alguma incompatibilidade, é possível ainda usar o recurso *MJI* (Model Java Interface) do *JPF*, disponível para extensão da ferramenta, ou identificar algum projeto já realizado na página do *JPF* que visa lidar com a biblioteca da função em questão.

Durante o desenvolvimento do sistema esse procedimento foi realizado para a validação do uso biblioteca `java.nio.ByteBuffer` que foi essencial para o sistema.

Nossa abordagem de verificação do sistema fizemos uso da distribuição padrão do JPF, que está presente no `jpf-core`. Nessa distribuição básica três propriedades são avaliadas.

Sendo elas as seguintes:

- `gov.nasa.jpf.jvm.NotDeadlockedProperty` - para todo estado não terminal, testa se ainda existe uma thread executável;
- `gov.nasa.jpf.jvm.NoAssertionViolatedProperty` - testa se alguma asserção/expressão é violada;
- `gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty` - testa se alguma exceção não foi tratada dentro da aplicação.

Além dessas três propriedades já pré-definidas temos três formas de busca na árvore de estados. Esses heurísticas são:

- RANDOM SEARCH (gov.nasa.jpf.search.RandomSearch) : busca aleatórizada;
- DFS SEARCH (search.class = gov.nasa.jpf.search.DFSearch): busca em profundidade;
- BFS SEARCH (search.class = .search.heuristic.BFSearch): busca em largura.

Todas as formas de busca são corretas, e no nosso caso as diferenças são quase irrelevantes na questão de performace conforme podemos perceber nos resultados obtidos no experimento realizado onde a mesma execução, com busca pelas mesmas propriedades padrões foi realizada no mesmo ambiente com as diferentes heurísticas. Abaixo os resultados estão ilustrados nas Figuras 5.4, 5.2, 5.3 e resumidos na tabela 5.1:

```
Camilas-MacBook-Air:src cachutti$ java -jar /Users/cachutti/projects/jpf-core/build/RunJPF.jar +shell.port=4242
/Users/cachutti/NetBeansProjects/SWPDC/src/swpdc/SWPDC.jpf
JavaPathfinder v6.0 (rev ${version}) - (C) RIACS/NASA Ames Research Center


============================================== system under test
application: swpdc/SWPDC.java

============================================== search started: 10/24/13 12:41 PM

============================================== results
no errors detected

============================================== statistics
elapsed time:       00:00:01
states:             new=38, visited=0, backtracked=38, end=0
search:             maxDepth=38, constraints hit=0
choice generators:  thread=38 (signal=0, lock=1, shared ref=0), data=0
heap:               new=2905, released=707, max live=2245, gc-cycles=37
instructions:       191502
max memory:         81MB
loaded code:        classes=250, methods=3533

============================================== search finished: 10/24/13 12:41 PM
```

Figura 5.1: Resultado DFS

```
JavaPathfinder v6.0 (rev ${version}) - (C) RIACS/NASA Ames Research Center


============================================================ system under test
application: swpdc/SWPDC.java

============================================================ search started: 10/24/13 3:46 PM
paths = 1

============================================================ results
no errors detected

============================================================ statistics
elapsed time:       00:00:01
states:             new=38, visited=0, backtracked=0, end=0
search:             maxDepth=0, constraints hit=0
choice generators:  thread=38 (signal=0, lock=1, shared ref=0), data=0
heap:               new=2902, released=705, max live=2244, gc-cycles=37
instructions:       191502
max memory:         81MB
loaded code:        classes=250, methods=3533

============================================================ search finished: 10/24/13 3:46 PM
```

Figura 5.2: Resultado BFS

```
JavaPathfinder v6.0 (rev ${version}) - (C) RIACS/NASA Ames Research Center


============================================================ system under test
application: swpdc/SWPDC.java

============================================================ search started: 10/24/13 3:30 PM

============================================================ results
no errors detected

============================================================ statistics
elapsed time:       00:00:01
states:             new=38, visited=0, backtracked=38, end=0
search:             maxDepth=38, constraints hit=0
choice generators:  thread=38 (signal=0, lock=1, shared ref=0), data=0
heap:               new=2902, released=705, max live=2244, gc-cycles=37
instructions:       191502
max memory:         81MB
loaded code:        classes=250, methods=3533

============================================================ search finished: 10/24/13 3:30 PM
```

Figura 5.3: Resultado Random

Diante tabela 5.1 fica evidente que a opção de um pelo outro é indiferente. No nosso caso optamos pela heurística DFS por está já ser a opção default do sistema. Observamos,

Tabela 5.1: Comparação das heurísticas de busca.

quantidade de...	DFS	BFS	Random
estados novos	38	38	38
profundidade máxima	38	0	38
thread	38	38	38
novos heaps	2905	2902	2902
instruções	191502	191502	191502
memória máxima	81MB	81MB	81MB
classes	250	250	250
métodos	3533	3533	3533

então o resultado de cada uma das execuções para cada uma das propriedades. Aqui dividimos em dois casos: propriedades que não foram violadas e propriedades violadas. Para as propriedades padrões temos a seguinte conjuntura:

- propriedade(s) não violadas:
  - `gov.nasa.jpf.jvm.NotDeadlockedProperty`
  - `gov.nasa.jpf.jvm.NoAssertionViolatedProperty`

Aqui a mensagem recebida como resposta final da busca do `jpf-core` é: `no errors detected`. O relatório obtido está ilustrado na Figura 5.4.

```
Camilas-MacBook-Air:src cachutti$ java -jar /Users/cachutti/projects/jpf-core/build/RunJPF.jar +shell.port=4242
/Users/cachutti/NetBeansProjects/SWPDC/src/swpdc/SWPDC.jpf
JavaPathfinder v6.0 (rev ${version}) - (C) RIACS/NASA Ames Research Center


============================================================ system under test
application: swpdc/SWPDC.java

============================================================ search started: 10/24/13 12:41 PM

============================================================ results
no errors detected

============================================================ statistics
elapsed time:       00:00:01
states:             new=38, visited=0, backtracked=38, end=0
search:             maxDepth=38, constraints hit=0
choice generators:  thread=38 (signal=0, lock=1, shared ref=0), data=0
heap:               new=2905, released=707, max live=2245, gc-cycles=37
instructions:       191502
max memory:         81MB
loaded code:        classes=250, methods=3533

============================================================ search finished: 10/24/13 12:41 PM
```

Figura 5.4: Exemplo de resultado positivo

- propriedade(s) violadas:

– `gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty`

Nesses casos, o JPF apresenta um resultado detalhado da localização do erro encontrado. Isso está ilustrado na imagem 5.5 abaixo onde vemos o relatório gerado para o sistema em teste, o SWPDC:

```
JavaPathfinder v6.0 (rev ${version}) - (C) RIACS/NASA Ames Research Center


====================================================== system under test
application: swpdc/SWPDC.java

====================================================== search started: 10/24/13 1:13 PM

====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.NoSuchMethodException: Calling java.io.FileOutputStream.<init>(Ljava/io/File;Z)V
    at java.io.FileWriter.<init>(FileWriter.java:90)
    at suporte.Comunicador.<init>(Comunicador.java:33)
    at suporte.Comunicador.instanciar(Comunicador.java:46)
    at suporte.Iniciador.iniciar(Iniciador.java:19)
    at swpdc.SWPDC.main(SWPDC.java:26)


====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,lockCount:0,suspendCount:0}
  call stack:
    at java.io.FileWriter.<init>(FileWriter.java:90)
    at suporte.Comunicador.<init>(Comunicador.java:33)
    at suporte.Comunicador.instanciar(Comunicador.java:46)
    at suporte.Iniciador.iniciar(Iniciador.java:19)
    at swpdc.SWPDC.main(SWPDC.java:26)

====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty "java.lang.NoSuchMethodException: Calling java.io
.F..."

====================================================== statistics
elapsed time:       00:00:01
states:             new=38, visited=0, backtracked=0, end=0
search:             maxDepth=38, constraints hit=0
choice generators:  thread=38 (signal=0, lock=1, shared ref=0), data=0
heap:               new=2902, released=705, max live=2244, gc-cycles=37
instructions:       191502
max memory:         81MB
loaded code:        classes=250, methods=3533

====================================================== search finished: 10/24/13 1:13 PM
```

Figura 5.5: Exemplo de resultado onde um erro foi encontrado

Aqui fica evidente que o problema está na linha 33 da classe Comunicador.java, conforme relatório apresentado.

Sabendo disso, a segunta etapa da nossa verificação, que foi realizada por cenários, teve de ser adaptada para não fazer uso de qualquer biblioteca de entrada e saída sem perder de vista a sistema original desenvolvido. Sendo assim, o primeiro passo pra o desenvolvimento das propriedades sobre métodos e classes, foi o estudo dos nossos cenários para depreender quais as entidades que deveriam ser verificadas para poder aferir a corretude dos requisitos. O resultado é apresentado na tabela 5.2.

As entidades listadas acima só poderiam ser verificadas por meio do desenvolvimento de proriedades JPF. Tais propriedades foram desenvolvidas usando a ideia de Listeners.

Os listeners permitem a criação de classes Java que são notificadas quando determinados eventos acontecem, como por exemplo quando determinada instrução foi executada. A principal classe utilizada para a implementação das propriedades foi a `gov.nasa.jpf.ListenerAdapter` que implementa todos os métodos das `SearchListener` e `VMListener`.

Eles são, talvez, o mais importante mecanismo de extensão do JPF. Fornecem uma maneira de observar, interagir e ampliar a execução JPF com classes próprias, uma vez que os listeners são configurados dinâmicamente em tempo de execução, não exigindo nenhuma

Tabela 5.2: Entidades relevantes para cada requisito dos cenários

Código	**Adaptação**	**Casos de teste**
SRS001	ADAPTADA - O SWPDC será startado e verificará a situação do simulador. Este já deve ter sido inicializado e se encontrar no estado NOMINAL para que a troca de comando e dados seja iniciada.	Iniciador, Verificador e Comunicador
SRS002	ADAPTADA - O SWPDC só começa depois de dar um POST no comando para verificar o estado do PDC (INFORMAR MODO DE OPERAÇÃO) e só vai começar a "usá-lo"se a resposta for a de que está tudo corretamente inicializado - ANÁLOGO AO REQUISITO SRS001 ADAPTADO	Iniciador, Verificador e Comunicador
SRS003	MANTIDA - Se o PDC não apresentar qualquer problema irrecuperável , após o processo de iniciação, o PDC entra automaticamente no modo de operação de NOMINAL	Verificador
POCP01	ADAPTADA - O PDC só pode responder às solicitações (comandos) de SWPDC após o PDC ter sido energizado por pelo menos 1 minuto. Se o SWPDC envia comandos em menos de 1 minuto, ele não recebe qualquer resposta do PDC.	Limitação do JPF
RB001	ADAPTADA - O SWPDC enviará INFORMAR ESTADO para PDC	Comunicador
SRS004	MANTIDA - O SWPDC deve esperar pelo menos 600 segundo antes de pedir dados de housekeeping para o PDC	Limitação do JPF
SRS005	ADAPTADA - A transmissão de 12 amostras deve ser realizada a cada requisição.	BufferSimples
SRS009	MANTIDA - O SWPDC devem distinguir entre processos de Power On / Reset	Verificador
RB005	Não se aplica	-

modificação no núcleo do JPF. São executados no mesmo nível do núcleo do JPF, então não há qualquer limitação adicional para os listeners que não exista também para o JPF como um todo.

A sua relação com o verificador de modelos está ilustrado na Figura 5.6.

O princípio geral é simples: JPF fornece uma implementação no padrão Observer que notifica casos de ocorrência registrados sobre determinados eventos em nível de VM. Essas notificações abrangem um amplo esspectro de operações JPF, desde de eventos de baixo nível como `instructionExecuted` até eventos de alto nível como `searchFinished`.

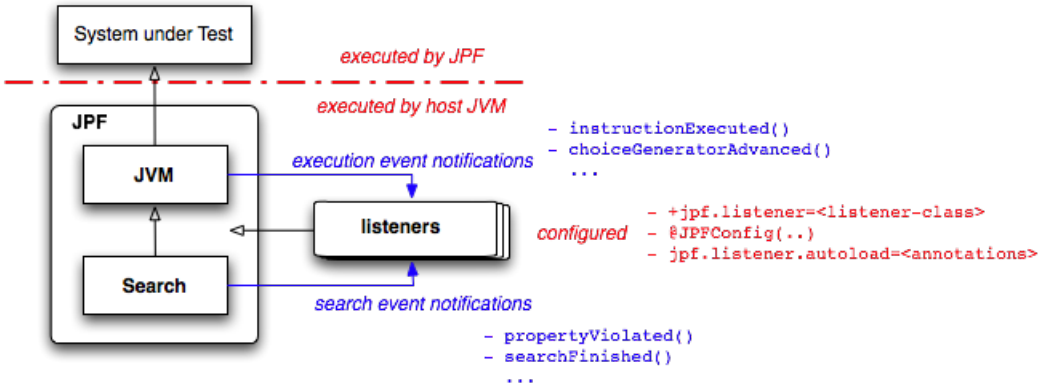

Figura 5.6: Esquema geral da relação dos listeners com o JPF cores.

Existem duas interfaces básicas para configuração e desenvolvimento de Listeners , dependendo dos eventos que se quer perquisar: SearchListeners e VMListeners. Uma vez que essas interfaces são muito grandes, e muitas vezes os Listeners não precisam implementar um escopo tão grande para desenvolver uma só propriedade, existem os Adaptaders, ou seja, implementadores que contêm todas as definições de métodos necessários, sendo assim, Listeners que estendem esses adaptadores, só têm que substituir os métodos de notificação que estão interessados dentro da sua própria classe. Devido a essa facilidade os adapters são usados na maioria das implementações de listeners. A Figura 5.7 ilustra a relação entre os tipos de listeners disponíveis:

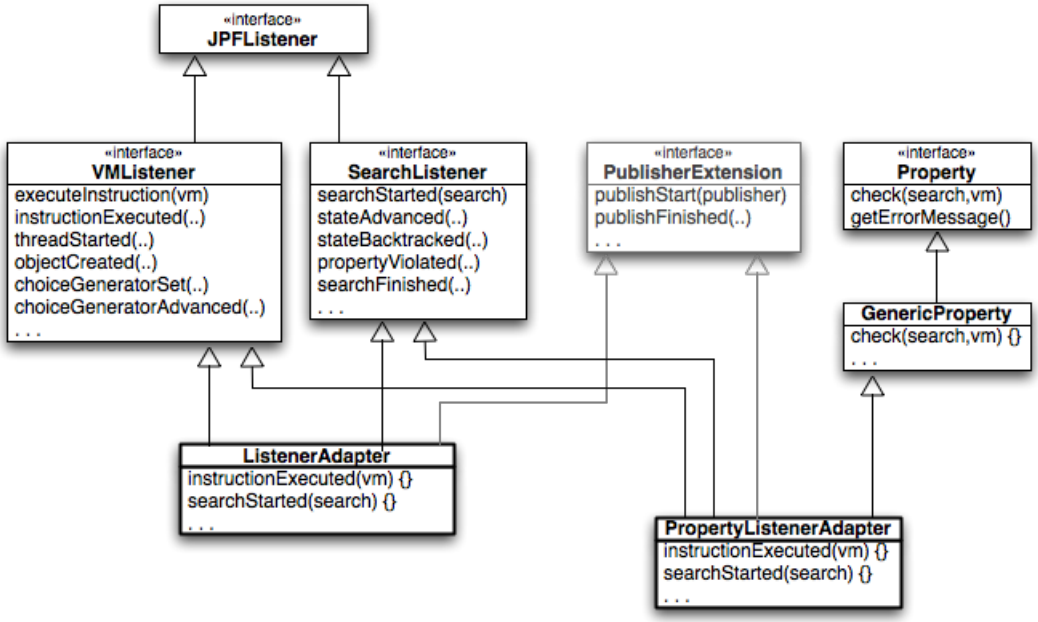

Figura 5.7: Tipos de Listeners.

Nos valendo também da facilidade de implementação com uso de adapters todas as propriedades desenvovlidas estendem `PropertyListenerAdapter` e a ideia geral que permeia todas as propriedades desenvolvidas é verificar os métodos essenciais das entidades envolvida no que diz respeito as subfunções que cada um desses métodos faz uso.

Analisando uma das propriedades implementadas para esclarecimento dos procedimentos práticos envolvidos na produção de uma propriedade, vamos detalhar a lógica utilizada para a produção da propriedade `InstanciarVerificadorCorrectlyProperty`. Todas as outras propriedades estão em anexo a este relatório.

A implementação da propriedade citada é o seguinte:

```
1  public class InstanciarVerificadorCorrectlyProperty
2                                          extends PropertyListenerAdapter {
3    private int count = 0;
4
5    @Override
6    public boolean check(Search search, JVM vm) {
7      return (count > 0);
8    }
9
10   @Override
11   public void instructionExecuted(JVM vm) {
12     Instruction instruction = vm.getLastInstruction();
13     MethodInfo mi = instruction.getMethodInfo();
14     ThreadInfo ti = vm.getLastThreadInfo();
15     if (!ListenerUtils.filter(instruction)) {
16       System.out.println(">> "+ instruction.getSourceLine().trim());
17       System.out.println(mi.getFullName());
18       if (instruction.getSourceLine().trim().contains("Verificador"))
19         count ++;
20     }
21   }
22
23   @Override
24   public String getErrorMessage() {
25     return "Property Violation: count = " + count;
26   }
27 }
```

Nesta propriedade implementamos três funções da interface `PropertyListenerAdapter`, sendo elas e seus respectivos comportamentos:

- `public boolean check (Search search, JVM vm)`: verificador booleano de violação de propriedade. Se não ouve qualquer violação retorna `TRUE`, caso contrário retorna `FALSE`, pois houve violação;
- `public void instructionExecuted (JVM vm)`: este método é o principal para a execução e verificação da corretude da função. Este olha para as instruções executadas (`Instruction instruction = vm.getLastInstruction()`), recupera as informações dessa instrução (`MethodInfi mi = instruction.getMethodInfo()`) e da

thread em que ela foi chamada (`ThreadInfo ti = vm.getLastThreadInfo()`). Com essas informações, é possível verificar se algum método tem o nome do método que queremos ter certeza que foi executado durante um fluxo do programa principal proposto. Essa informação é passada no último `if` da função na seguinte linha de código `instruction.getSourceLine().trim().contains("nome_da_função")`, onde `nome_da_função` é uma string que será buscada no nome de todas as instruções executadas, para que possamos afirmar se a mesma foi ou não chamada.

- `public String getErrorMessage():` se houve erro, recupera qual foi esse erro e imprime uma mensagem no relatório gerado para informar o usuário.

Outro comportamento adotado pra viabilizar a verificação do SWPDC, foi o desenvolvimento de dois `mains` voltados para a verificação dos componentes de suporte e dados. Estes encapsularam a verificação de cada um deles e também não fizeram uso de qualquer biblioteca de entrada e saída. Os mains alternativos produzidos estão em anexo e servem apenas como um organizador de chamadas de função. Podemos ver no código abaixo do `main` alternantivo produzido para a verificação do comunicador, do verificador e iniciador:

```
1  package swpdc;
2  import suporte.Comunicador;
3  import suporte.Iniciador;
4  import suporte.Verificador;
5
6  public class MainParaVerificacaoSuporte {
7      static Comunicador comunicador;
8      static Verificador verificador;
9      static Iniciador iniciador;
10
11     public static void main(String[] args){
12         iniciador = new Iniciador();
13         verificador = Verificador.instanciar();
14         comunicador = Comunicador.instanciar();
15         iniciador.iniciar();
16     }
17 }
```

Observando o código acima é fácil perceber que este apenas chama as funções de inicialização dos componentes de suporte para que a verificação possa ser verificado.

As propriedades desenvolvidas foram as seguintes apresentadas na tabela 5.3:

Juntando as duas tabelas apresentadas acima, podemos alcançar o seguinte mapeamento de requisitos dos cenários em propriedades JPF apresentado na tabela 5.4:

Tabela 5.3: Adaptação dos cenários para produção das propriedades no JPF

ID	Propriedade
P01	IniciarCorrectlyProperty
P02	InstanciarVerificadorCorrectlyProperty
P03	InstanciarComunicadorCorrectlyProperty
P04	ConstructedBufferSimplesCorrectlyProperty
P05	AllocatedCorrectlyProperty
P06	InserirTemperaturaCorrectlyProperty
P07	RecuperarTemperaturaCorrectlyProperty

Com as propriedades implementadas e verificadas podemos afirmar que os requisitos dos cenários 17 e 50 estão verificados. Os mains alternativos produzidos e todas as propriedades desenvolvidas estão em anexo.

Para realizar a verificação propriamente dita foram criados dois arquivos: `dados.jpf` e `suporte.jpf`, que são os responsáveis por ligar os listeners usados para verificar os respectivos componentes. Ambos podem ser vistos abaixo:

- `dados.jpf`

```
1 target = swpdc.MainParaVerificacaoDados
2 cg.enumerate_random = true
3
4 listener = propriedades.AllocatedCorrectlyProperty
5 listener = propriedades.ConstructedBufferSimplesCorrectlyProperty
6 listener = propriedades.InserirCorrectlyProperty
7 listener = propriedades.RecuperarCorrectlyProperty
```

- `suporte.jpf`

```
1 target = swpdc.MainParaVerificacaoSuporte
2 cg.enumerate_random = true
3
4 listener = propriedades.IniciarCorrectlyProperty
5 listener = propriedades.InstanciarComunicadorCorrectlyProperty
6 listener = propriedades.InstanciarVerificadorCorrectlyProperty
```

Os acima definem o programa alvo da verificação no argumento `target`, permite a criação de dados aleatórios para as verificação padrões já abordadas acima e liga os listener que desejamos nos argumentos `listener`.

- Dificuldades

  A principal dificuldade encontrada durante a verificação do sistema foi que em alguns casos verificar uma funcionalidade em si foi mais complexo que a implementação da mesma, que para ser desenvolvida levou alguns minutos e desenvolver uma propriedade para verificá-la levou algumas horas.

Tabela 5.4: Adaptação dos cenários para produção das propriedades no JPF

Código	**Adaptação**	**Casos de teste**
SRS001	ADAPTADA - O SWPDC será startado e verificará a situação do simulador. Este já deve ter sido inicializado e se encontrar no estado NOMINAL para que a troca de comando e dados seja iniciada.	P01; P02; P03
SRS002	ADAPTADA - O SWPDC só começa depois de dar um POST no comando para verificar o estado do PDC (INFORMAR MODO DE OPERAÇÃO) e só vai começar a "usá-lo"se a resposta for a de que está tudo corretamente inicializado - ANÁLOGO AO REQUISITO SRS001 ADAPTADO	P01; P02; P03
SRS003	MANTIDA - Se o PDC não apresentar qualquer problema irrecuperável , após o processo de iniciação, o PDC entra automaticamente no modo de operação de NOMINAL	P02
POCP01	ADAPTADA - O PDC só pode responder às solicitações (comandos) de SWPDC após o PDC ter sido energizado por pelo menos 1 minuto. Se o SWPDC envia comandos em menos de 1 minuto, ele não recebe qualquer resposta do PDC.	Limitação do JPF
RB001	ADAPTADA - O SWPDC enviará INFORMAR ESTADO para PDC	P03
SRS004	MANTIDA - O SWPDC deve esperar pelo menos 600 segundo antes de pedir dados de housekeeping para o PDC	Limitação do JPF
SRS005	ADAPTADA - A transmissão de 12 amostras deve ser realizada a cada requisição.	P04; P05; P06; P07
SRS009	MANTIDA - O SWPDC devem distinguir entre processos de Power On / Reset	P02
RB005	Não se aplica	-

Outra dificuldade que surge nos primeiros contatos com a ferramenta JavaPathfinder é a necessidade de conhecimento, ainda que superficial, da estrutura interna do JPF para realizar qualquer verificação mais simples. Tal necessidade existe pela grande flexibilidade da ferramenta, porém pode causar um resistência na sua adoção pelo mercado de software, que verá esse tempo de adaptação como uma impossibilidade de uso prático e real da ferramenta.

- Benefícios

A verificação e o desenvolvimento de propriedades usando JPF trouxe um grande ganho de qualidade para o software desenvolvido com a garantia de corretude das entidades centrais do sistema por meio da verificação de modelos.

Outro benefício bastante grande foi o aprendizado do funcionamento e estrutura da ferramenta JPF, que tem sido usada e desenvolvida pela comunidade científica, porém ainda não em larga escala e em softwares maiores como o desenvolvido nesse trabalho.

5.2 Análise dos Resultados e Conclusões

No desenvolvimento desse trabalhou muito se buscou por referências bibliográficas sobre, principalmente, desenvolvimento formal de sistemas reais que era um dos desafios da proposta e a conclusão que chegamos é que esse tipo de persquisa ainda é incipiente na literatura.

Outra defazagem na literatura científica é a avaliação do JPF dentro do contexto de verificação formal de programas. Esta se fazia necessária e trazê-la nesse trabalho mostrou-se um grande avanço. Percebe-se que o JPF pode ser usado em projetos grandes e complexos, mas ainda é complicado, uma vez que a sua adequação e capacidades ainda não estão totalmente definida, por exemplo não temos uma lista de bibliotecas externas que encontramos suporte no verificiador. Neste trabalho analisamos o suporte da biblioteca `java.nio.ByteBuffer` e das `java.io.BufferedWriter; java.io.File; java.io.FileWriter`, a primeira se mostrou adequada para uso com o JPF, já as outras não.

Uma grande colaboração científica veio também pela exposição da dificuldade da verificacão e um método de realização tanto pra este fim como para validação, trazendo um estudo do caso que até então não existia, onde o sistema a ser validado e verificado é real e tem uma escopo maior.

É necessário deixar evidente ainda que a combinação de técnicas de verificação e validação exploradas aqui provêem ao software mais qualidade se comparada ao uso de somente uma delas. Sendo assim, verificação formal não substitui totalmente a validação e a simulação, assim como estas não substituem aquela.

As ferramentas e técnicas existentes de verificação ou validação, algumas delas analisada neste trabalho, ainda não são adequadas para o uso da grande comunidade de desenvolvedores. São ferramentas pouco intuitivas e têm um grande overhead de aprendizado que por vezes acaba dificultando a sua adoção. A verificação formal de modelos não é uma tarefa trivial e uma ferramenta que lide com isso também não é simples, mas sabendo da importância de produção de softwares com maior qualidade, a verificação formal deve estar ao alcance de qualquer projetista.

5.3 Atividades futuras

O projeto é bastante rico no ponto de vista teórico e prático. Dele será possível desenvolver diversos outros trabalhos e atividades. Algumas delas são:

- Desenvolvimento de uma Simulador de Eventos espaciais ainda mais completo com inconsistências reais de dados adquiridos por canais analógicos de sensoriamento.
- Implementação de todo o sistema SWPDC, acompanhada de verificação e validação do mesmo.
- Estudar e atacar as limitação encontradas no JPF durante o desenvolvimento desse trabalho e colaborar com um software livre e entregar para a comunidade uma ferramenta completa, robusta e de qualidade para verificação de modelos em programas Java.
- Estudar e usar outros projetos externos do do JavaPathfinder, por exemplo o jpf-statechart que se aplica muito bem no contexto desse trabalho que já tem alguns requisitos mapeados em Statecharts.

Parte II

PARTE SUBJETIVA

Capítulo 6

Aprendizado

6.1 Desafios e frustrações

O maior desafio de todos era conseguir produzir um trabalho que ao mesmo tempo que fosse bastante formal, para que eu pudesse aprender mais sobre uma área que não teria contato nas disciplinas oferecidas no curso, fosse também prático. Na minha visão esse desafio foi muito bem vindo e conseguimos produzir algo que alcançou as expectativas formais e práticas que tinha, uma fez que passamos por todos os passos do desenvolvimento de um software de missão crítica, que envolve bastante teória, mas tem uma grande carga prática uma vez que implementamos um produto de software.

Outro grande desafio foi ter que aplicar conhecimentos das mais diversas áreas no mesmo projeto e tudo aquilo que muitas vezes só tinha sido visto em algum livro teve que se tornar realidade em pouco menos de um ano. Acredito que este seja o grande desafio para a maioria dos estudantes que muitas vezes se atêm somente aos estudos e vêm pouca aplicação prática de grande parte do conteúdo coberto durante o bacharelado. Também estou confiante para afirmar que esse desafio foi bastante importante e consegui tirar proveito deste para o meu desenvolvimento acadêmico.

Já as frustrações ficaram principalmente por conta de uma ferramenta com papel central no desenvolvimento desse trabalho: o JavaPathfinder. Por vezes descobrir o que JPF de fato estava fazendo se mostrava uma tarefa impossível. Apesar de termos acesso ao código e este ter mecanismos de extensão foi bastante complicado entender sua estrutura, exigindo muitos testes, muitas horas de dedicação e vários pedidos de ajuda para o grupo de verificação formal do Departamento de Ciência da Computação do Instituto de Matemática e Estatística.

Outra frustração deste trabalho foi ter contato com uma especificação bastante completa e consistente de um software robusto e sair implementando do zero uma parcela bastante reduzida do que foi implementado no software real. Sabendo das dificuldade, com o tempo essa frustração acabou se tornando um desafio, pois nos fez ter que entender o software por inteiro para ter clareza ao escolher qual seria o escopo definido para ser implementado que tivesse sentido prático e científico e também trouxesse pelo menos um componente

completo do ponto de vista funcional.

Com a conclusão desse trabalho juntamente com a conclusão do curso de bacharelado em Ciência da computação tenho certeza que todos os desafios valeram a pena e as frustações foram alvo de grande aprendizado profissional e pessoal.

6.2 Disciplinas cursadas relevantes para o desenvolvimento do TCC

O curso como um todo me deu todas as ferramentas necessárias para formação de um pensamento lógico. Além disso me ensinou a aprender conteúdos novos com alguma facilidade pela minha formação de base sólida e não somente informativa. Dentre todas as mais de 40 matérias cursadas nesses quatro anos de curso as diretamente relevantes para o desenvolvimento desse trabalho foram:

- MAC0332 - Engenharia de Software

  Como essa disciplina também teve um projeto de maior escala, foi uma oportunidade para experimentar o desenvolvimento de um software com código legado e aplicar conceitos como Padrões de Projeto.

- MAC0242 - Laboratório de Programação II

  Primeiro contato com programação JAVA e um projeto maior.

- MAC0338 - Análise de Algoritmos

  Matéria que me deu base teórica para o desenvolvimento de um software eficiente.

- MAC0342 - Laboratório de Programação Extrema

  Matéria onde tive a oportunidade de aprender sobre o desenvolvimento de testes e fui bastante exposta a conceitos de limpeza de código.

- MAC0239 - Métodos Formais em Programação

  Introdução de linguagem lógica, provas de corretude e especificação formal e é foi o interesse por essa matéria que me envolvi no meu projeto de iniciação científica que posteriormente se tornou uma forte influência para o desenvolvimento desse trabalho.

6.3 Observações sobre a aplicação real de conceitos estudados

Acredito que o currículo proposto para o curso de Ciência da Computação do IME - USP é bastante abrangente e completo, porém sempre podemos melhorar. Na minha visão conceitos bastante úteis e imprescindíveis são pouco explorados no mesmo. Sendo os principais citados a seguir:

- Nenhuma matéria obrigatória da graduação aborda qualidade de software e seus tópicos afins. Durante a minha graduação apenas uma matéria me exigiu produção de testes para os produtos de softwares entregues (Laboratório de Programação Extrema), e esta matéria era uma optativa eletiva, sendo assim, alunos podem sair da graduação sem ter escrito uma única linha de teste para um software desenvolvido com fins acadêmicos.

- Programação Orientada à Objetos deveria ser obrigatória, assim como Programação Funcional Comtemporânea devido a importância de se ter conhecimento de programação em ambos os paradigmas.

- Durante a graduação nunca fui exposta a uma especificação formal de software em qualquer matéria, somente tive contato com isso durante a iniciação científica. Acredito que tanto a especificação em si de um software, quanto o desenvolvimento com base nela deveriam ser explorado durante a graduação pela sua aplicabilidade na produção de software real.

6.4 Próximos passos

A conclusão do curso de graduação e o desenvolvimento de diversas atividades paralelas como estágio, iniciação científica e monitorias me apresentaram diversas possibilidades. Acredito que hoje o passo mais natural, dado que quero continuar estudando é ingressar na pós-graduação e muito possivelmente continuar estudando temas que envolvam qualidade de software.

Além disso, trabalhar com desenvolvimento de software, ainda que por pouco tempo, no mundo real me trouxe bastante conhecimento e pude perceber que é possível aliar estudo e trabalho de maneira que os dois possam ser mutuamente beneficiados.

Capítulo 7

Agradecimentos

Agradecimentos a todos aqueles que acompanharam meu crescimento acadêmico colaborando comigo durante minha iniciação científica e desenvolvimento deste trabalho de conclusão de curso. Agradecimentos especiais ao Alexandre Locci e a Simone Hanazumi pelos esclarecimentos e apoio durante todo o processo. Obrigada professora Ana Cristina Vieira de Melo não só pela orientação atenciosa da minha pesquisa e trabalho de conclusão de curso, como também pelos discussões sobre minha formação, carreira acadêmica e profissional.

Agradeço os meus colegas de Instituto e Atlética, sem vocês o caminho até aqui não teria sido o mesmo. Sem vocês talvez não tivesse aprendido o que é de fato fazer parte de um time e ter muito orgulho disso. Obrigada meninas do volêi IME - USP, que me ensinaram muito mais que voleibol.

Agradeço também a minha família e amigos pelo constante apoio que sempre me deram e pela possibilidade de poder dividir minhas angústias e sucessos com vocês.

Obrigada a todos os professores e funcionários do Instituto de Matemática e Estatística da Universidade de São Paulo que me proporcionaram muito aprendizado e sempre confiaram em mim.

Referências Bibliográficas

[1] Aeronautics, AIAA(American Institute of e Astronautics. `http://www.javolution.org/` Acessada em Set. de 2013.

[2] Amir Pnueli, Zohar Manna e: *The temporal logic of reactive and concurrent systems*. 1992.

[3] Ammann, Paul e Jeff Offutt: *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1ª edição, 2008, ISBN 0521880386, 9780521880381.

[4] Baier, Christel e Joost Pieter Katoen: *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008, ISBN 026202649X, 9780262026499.

[5] Bertolino, Antonia: *Software Testing Research and Practice*. Em *10th International Workshop on Abstract State Machines*, páginas 1–21, 2003.

[6] Board, Inquiry: *Ariane 5 - Flight 501 Failure - Full Report*, 1996. `http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html`. Acesso em: Jul. 2012.

[7] Boehm, Barry W.: *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981, ISBN 0138221227.

[8] Boehm, Barry W.: *Software Engineering Economics*. Prentice Hall PTR, 1981.

[9] Chaim, Marcos Lordello: *Poke tool: uma ferramenta para suporte ao teste estrutural de programas baseados em anÃąlise de fluxo de dados*. 1991.

[10] Coverlipse. Disponível em: `http://coverlipse.sourceforge.net/` Acesso em: Jul. 2013.

[11] Delamaro, MÃąrcio Eduardo, JosÃl' Carlos Maldonado e Mario Jino: *IntroduÃğÃčo ao Teste de Software*. Elsevier, 2007.

[12] Dijkstra, Edsger W.: *Structured programming*. capítulo I: Notes on structured programming, páginas 1–82. Academic Press Ltd., London, UK, UK, 1972, ISBN 0-12-200550-3. `http://dl.acm.org/citation.cfm?id=1243380.1243381`.

[13] Dwyer, Matthew B., George S. Avrunin e James C. Corbett: *Patterns in property specifications for finite-state verification.* Em *Proceedings of the 21st international conference on Software engineering*, ICSE 99, páginas 411–420, New York, NY, USA, 1999. ACM, ISBN 1-58113-074-0. http://doi.acm.org/10.1145/302405.302672.

[14] Elaine J. Weyuker, Sandra Rapps e: *Data flow analysis techniques for test data selection.* ICSE 82: Proceedings of the 6th international conference on Software engineering, 1982.

[15] Elaine J. Weyuker, Sandra Rapps e: *Selecting software test data using data flow information.* 1985.

[16] FeaVer: *Disponível em: http://cm.bell-labs.com/cm/cs/what/feaver/index.html. Acesso em: Jul. 2013.*

[17] Francez, Nissim: *Program Verification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992, ISBN 0201416085.

[18] Francez, Nissim: *Program Verification.* 1992.

[19] Fátima Matiello-Francisco, Valdivino Alexandre de Santiago Jr., Ana Maria Ambrósio Maria de: *Verificação e Validação na terceirização de Software embarcado em aplicações espaciais.* V Simpósio Brasileiro de Qualidade de Software, 2006.

[20] Gosling, James, Bill Joy, Guy Steele e Gilad Bracha: *The Java Language Specification.* Addison-Wesley, 2005.

[21] Hoare, C. A. R.: *An axiomatic basis for computer programming.* ACM, 1969.

[22] Holzmann, Gerard J.: *Design and validation of computer protocols.* Prentice-Hall, Inc., 1991.

[23] Holzmann, Gerard J.: *The model checker spin.* 1997.

[24] James C. Corbett, Matthew B. Dwyer, John Hatcliff Shawn Laubach Corina S. Pa sareanu Robby e Hongjun Zheng: *Bandera : Extracting finite-state models from java source code.* 2000.

[25] Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/. Acesso em: Jul. 2012.

[26] Jim Davies, Jim Woodcock e: *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc, 1996.

[27] Jones, Cliff B.: *Systematic software development using VDM (2nd ed.).* Prentice-Hall, 1990.

[28] Joost-Pieter Katoen, Christel Baier e: *Principles of Model Checking (Representation and Mind Series)*. 2008.

[29] José Carlos Maldonado, Marcos Lordello Chaim, e Mário Jino: *Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos*. 1989.

[30] JUnit. `http://pt.wikipedia.org/wiki/JUnit`. Acesso em: Jul. 2013.

[31] Lamport, Leslie: *The temporal logic of actions*.

[32] Matthew B. Dwyer, John Hatcliff e: *Using the bandera tool set to modelcheck properties of concurrent java software*. CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory, páginas 39 – 58, 2001.

[33] Melo, Ana Cristina Vieira De, Flávio Soares Correa Da Silva e Marcelo Finger: *Lógica para Computação*. Thomson Learning, 2006, ISBN 8522105170.

[34] Márcio Eduardo Delamaro, José Carlos Maldonado, e Mário Jino: *Introdução ao Teste de Software*. 2007.

[35] Oracle: *Java Technology*. `http://www.oracle.com/br/technologies/java/index.html`. Acesso em: Jul. 2012.

[36] Pnueli, Amir: *The temporal logic of programs*. SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, páginas 46 – 57, 1977.

[37] Pressman, Roger S.: *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001, ISBN 0072496681.

[38] Păsăreanu, Corina S., Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person e Mark Pape: *Combining unit-level symbolic execution and system-level concrete execution for testing nasa software*. Em *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, páginas 15–26, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-050-0. `http://doi.acm.org/10.1145/1390630.1390635`.

[39] Santiago Júnior, Valdivino Alexandre de: *SOLIMVA: A Methodology for Generating Model-Based Test Cases from Natural Language Requirements and Detecting Incompleteness in Software Specifications*. 2011.

[40] Silva Xavier, Kleber da: *Estudo sobre redução do custo de testes através da utilização de verificação de componentes java com tratamento de exceções*. 2008.

[41] Sommerville, Ian: *Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8th edição, 2006.

[42] SPF. `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`. Acesso em: Jul. 2012.

[43] SPIN. Disponível em: `http://spinroot.com`. Acesso em: Jul. 2013.

[44] Ujma, Mateusz e Nastaran Shafiei: *jpf-concurrent: An extension of Java PathFinder for java.util.concurrent*. The Computing Research Repository (CoRR), abs/1205.0042, 2012.

[45] W. Visser, K. Havelund, G. Brat S. Park e F. Lerda: *Model checking programs*. Automated Software Engineering Journal, 2003.

[46] Woodcock, Jim, Peter Gorm Larsen, Juan Bicarregui e John Fitzgerald: *Formal methods: Practice and experience*. ACM Comput. Surv., 41(4):19:1–19:36, oct 2009, ISSN 0360-0300. `http://doi.acm.org/10.1145/1592434.1592436`.

[47] Wordsworth, J. B.: *Software engineering with B*. 1996.

Parte III

Anexos

Apêndice A

Modelos dos arquivos compartilhados

- `hitorico.txt`

```
1  [data] xx/xx/xx [hora] xx:xx:xx.xx emitiu comando: COMANDO
2
3  Exemplo:
4  10/09/2013 13:19:33.861 emitiu comando: INFORMAR ESTADO
```

- `instrucoes.txt`

```
1  COMANDO
2  COMANDO
3  COMANDO
4      .
5      .
6      .
7  COMANDO
8  COMANDO
9  COMANDO
10
11 - - - - - - - - - - - - - - - - -Exemplo:
12 COMECAR SIMULACAO
13 INFORMAR ESTADO
14 OBTER TEMPERATURA
15 OBTER RELOGIO
16 COMPENSAR DRIFT RELOGIO
17 OBTER TEMPERATURA
18 OBTER TEMPERATURA
19 OBTER TEMPERATURA
```

- `registroTemp.txt`

```
1
2  [numAm] ID-AMOSTRA [dados] temp 1, temp 2, ..., temp 12
3  Exemplo:
4  [numAm] 1 [dados] 11.5, 45.7, 40.0, 34.8, 39.0, 35.0,
5                    12.4, 10.7, 27.0, 09.0, 23.7, 19.7
```

```
6 [numAm] 2 [dados] 12.5, 15.7, 40.0, 34.8, 39.0, 35.0,
7                           12.4, 10.7, 27.0, 10.0, 23.7, 19.7
8 [numAm] 3 [dados] 13.5, 15.7, 40.0, 34.8, 39.0, 35.0,
9                           12.4, 10.7, 27.0, 10.0, 23.7, 19.7
```

- `RelatorioPacoteHK.txt`

```
1  Relatorio PacoteHK
2
3  [data] xx/xx/xxxx [hora] xx:xx:xx
4  [modoOpSWPDC] xxxxxxxx
5
6  Amostras de temperatura:
7  xx
8  xx
9  xx
10 xx
11 xx
12 xx
13 xx
14 xx
15 xx
16 xx
17 xx
18 xx
19 x Erros
20 Tempo de amostragem: xx:xx
21
22 Historico de Eventos
23 -
24 -
25 -
```

Apêndice B

Especificação do Simulador

- DADOS:
  - temperatura (ESSENCIAL) - (10 a 40 graus celsius)
  - relógio (horas/mins/segs/ms) - (relógio de 24 horas na precisão de ms).
  - estado (inicialmente só o nominal)
  - testes (DESEJÁVEL)
  - diagnóstico (OPCIONAL)
- COMANDOS:
  - OBTER TEMPERATURA ‘
  - OBTER RELÓGIO
  - INFORMAR MODO DE OPERAÇÃO (Os possíveis são: diagnóstico, segurança, iniciação e nominal)
  - RECUPERAR LOG
  - COMPENSAR DRIFT RELÓGIO (Pegar o relógio do software, ver a diferença entre eles, ver se algum procedimento ocorreria naquela diferença, se sim fazer de uma maneira que aquele procedimento não seja perdido, mas o relógio se iguale, se não houver qualquer problema basta igualá-los)
- ESPECIFICAÇÃO FUNCIONAL:
  1. O PDC_simulador deverá ter, em memória, informações que definam a configuração atual do subsistema de computação. Sendo elas:
     - informação se o PDC está ligando ou se está desligado.
     - modo de operação
     - Contadores de erros/log
     - Ponteiros de escrita e leitura associados ao gerenciamento de memória (infos sobre a quantidade de dados)

2. Tempo de amostragem dos canais analógicos de entrada é 10s (no nosso caso é o delta t de geração de dados)
3. SWPDC/PDC deve se preocupar em detectar e lidar com erros na memória, recepção incompleta (timeout) e valores fora do intervalo
4. O PDC deve operar em um intervalo de temperatura que varia de 10 graus celsius até 40 graus celsius, portanto, o SWPDC deverá medir temperaturas neste intervalo
5. O PDC possui apenas 2 modos de alimentação: desligado e potência normal
6. O PDC possuirá um relógio de 32 bits com resolução 1ms
7. Deve enviar o relógio para o SWPDC e ser capaz de sincronizá-lo se necessário.
8. Deve estar preparado para os seguintes comandos: OBTER TEMPERATURA, OBTER RELÓGIO, INFORMAR ESTADO, RECUPERAR LOG, COMPENSAR DRIFT RELÓGIO

Apêndice C

Implementação dos principais testes do sistema

- Testes da classe BufferSimples.java

```
1  package dados;
2
3  import java.nio.ByteBuffer;
4  import org.junit.After;
5  import org.junit.AfterClass;
6  import org.junit.Before;
7  import org.junit.BeforeClass;
8  import org.junit.Test;
9  import static org.junit.Assert.*;
10
11
12 /**
13  *
14  * @author Camila Achutti
15  */
16 public class BufferSimplesTest {
17
18     public BufferSimplesTest() {
19     }
20
21     @Test
22     public void testAlocarVazio() {
23         System.out.println("alocarVazio");
24         Object area = null;
25         byte tamanho = 0;
26         BufferSimples instance = new BufferSimples(area, tamanho);
27         instance.alocar(area, tamanho);
28         }
29
30     @Test
31     public void testAlocar() {
32         System.out.println("alocar");
```

```
33          Object area = null;
34          byte tamanho = (byte)100;
35          BufferSimples instance = new BufferSimples(area, tamanho);
36          instance.alocar(area, tamanho);
37          }
38
39      @Test
40      public void testEstaVazio() {
41          System.out.println("estaVazio");
42          Object area = null;
43          byte tamanho = 0;
44          BufferSimples instance = new BufferSimples(area, tamanho);
45          boolean expResult = true;
46          boolean result = instance.estaVazio();
47          assertEquals(expResult, result);
48          }
49
50      @Test
51      public void testInserir() {
52          int index = 0;
53          float dado = (float) 12.4;
54          byte tamanho = (byte)100;
55          Object area = null;
56          BufferSimples instance = new BufferSimples(area, tamanho);
57          boolean result = instance.inserir(dado, (byte)4, index);
58          assertTrue(result);
59        }
60
61
62      @Test
63      public void testTamanho() {
64          System.out.println("tamanho");
65          byte tamanho = (byte) 100;
66          BufferSimples instance = new BufferSimples(null, tamanho);
67          byte result = instance.tamanho();
68          assertEquals(tamanho, result);
69
70      }
71
72      @Test
73      public void testRecuperar() {
74          System.out.println("recuperar");
75          int index = 0;
76          float dado = (float) 19.4;
77          float dadoRecuperado;
78          byte tamanho = 100;
79          BufferSimples instance = new BufferSimples(null, tamanho);
80          instance.inserir(dado, tamanho, index);
81          dadoRecuperado = instance.recuperar(0);
82          assertTrue(dadoRecuperado - dado == 0);
83       }
```

```
84
85
86      @Test
87      public void testRemover() {
88          System.out.println("remover");
89          int index = 0;
90          float dado = (float)18.5;
91          byte tamanho = 100;
92          BufferSimples instance = new BufferSimples(null, tamanho);
93          BufferSimples expected = new BufferSimples(null, tamanho);
94          instance.inserir(dado, tamanho, index);
95          instance.remover(100);
96          assertTrue(instance.posicao() == expected.posicao());
97      }
98  }
```

- Testes da classe Iniciador.java

```
1   package suporte;
2
3   import org.junit.After;
4   import org.junit.AfterClass;
5   import org.junit.Before;
6   import org.junit.BeforeClass;
7   import org.junit.Test;
8   import static org.junit.Assert.*;
9   import static org.mockito.Mockito.*;
10
11  /**
12   *
13   * @author Camila Achutti
14   */
15  public class IniciadorTest {
16      private Comunicador comunicador;
17      private Verificador verificador;
18      private Iniciador iniciador;
19
20      public IniciadorTest() {
21      }
22
23      @Before
24      public void setUp() {
25          comunicador = mock(Comunicador.class);
26          verificador = mock(Verificador.class);
27
28          iniciador = new Iniciador(comunicador, verificador);
29          when(verificador.verificaStatusAlimentacaoLigado()).thenReturn(
30                                                                  true);
31
32      }
33
```

```
34      @Test
35      public void testIniciarFinallizaComSucesso() {
36          iniciador.iniciar();
37
38          verify(comunicador).emitirComando("COMEÇAR SIMULAÇAO");
39          verify(comunicador).guardarNoHistorico(
40                                              "PDC inicializado com sucesso.");
41          assertEquals(1, iniciador.m_relPOST);
42      }
43
44      @Test
45      public void testObterEstadoPDCNominal() {
46          when(verificador.verificaEstado()).thenReturn("nominal");
47
48          assertEquals(1, iniciador.obterEstadoPDC());
49
50          verify(comunicador).emitirComando("INFORMAR ESTADO");
51          verify(comunicador).guardarNoHistorico(
52          "Verificação de estado do PDC realizada com sucesso
53                                                  - status: nominal");
54          verify(comunicador).guardarNoHistorico(
55              "PDC está no estado NOMINAL. Inicialização bem sucedida");
56      }
57
58      @Test
59      public void testObterEstadoPDCDiferenteNominal() {
60          when(verificador.verificaEstado()).thenReturn("iniciacão");
61
62          assertEquals(0, iniciador.obterEstadoPDC());
63
64          verify(comunicador).emitirComando("INFORMAR ESTADO");
65          verify(comunicador).guardarNoHistorico(
66          "Verificação de estado do PDC realizada com sucesso
67                                                  - status: iniciacão");
68          verify(comunicador).guardarNoHistorico(
69              "PDC NÃO está no estado NOMINAL.");
70      }
71
72      @Test
73      public void testAtivarModuloDados() {
74          assertEquals(1, iniciador.ativarModuloDados());
75      }
76  }
```

- Testes da classe Verificador.java

```
1   package suporte;
2
3   import java.io.BufferedReader;
4   import java.io.IOException;
5   import org.junit.Before;
```

```
6  import org.junit.Test;
7  import static org.junit.Assert.*;
8  import static org.mockito.Mockito.*;
9
10 /**
11  *
12  * @author Camila Achutti
13  */
14 public class VerificadorTest {
15     private BufferedReader bufReaderRegistroTemp;
16     private BufferedReader bufReaderSysInfo;
17     private Comunicador comunicador;
18     private Verificador verificador;
19
20     @Before
21     public void setUp() {
22         bufReaderRegistroTemp = mock(BufferedReader.class);
23         bufReaderSysInfo = mock(BufferedReader.class);
24         comunicador = mock(Comunicador.class);
25
26         verificador = new Verificador(bufReaderRegistroTemp,
27                                       bufReaderSysInfo, comunicador);
28     }
29
30     @Test
31     public void testVerificaStatusAlimentacaoLigadoSucesso()
32                                                     throws IOException {
33         when(bufReaderSysInfo.readLine()).thenReturn("[status] ligado");
34
35         assertTrue(verificador.verificaStatusAlimentacaoLigado());
36     }
37
38     @Test
39     public void testVerificaStatusAlimentacaoDiferenteDeLigado()
40                                                     throws IOException {
41          when(bufReaderSysInfo.readLine()).thenReturn(
42                                                     "[status] desligado");
43
44          assertFalse(verificador.verificaStatusAlimentacaoLigado());
45     }
46
47     @Test
48     public void testVerificaStatusAlimentacaoDiferenteExcecao()
49                                                     throws IOException {
50         when(bufReaderSysInfo.readLine()).thenThrow(new IOException());
51
52         assertFalse(verificador.verificaStatusAlimentacaoLigado());
53     }
54
55     @Test
56     public void testVerificaEstadoQuandoLeModoOp()
```

```
57                                                      throws IOException {
58          when(bufReaderSysInfo.readLine()).thenReturn(
59                                                      "[modoOp] nominal");
60
61          assertEquals("nominal", verificador.verificaEstado());
62      }
63
64      @Test
65      public void testVerificaEstadoQuandoLeNull()
66                                                      throws IOException {
67          when(bufReaderSysInfo.readLine()).thenReturn(null);
68
69          assertEquals(null, verificador.verificaEstado());
70          verify(comunicador).guardarNoHistorico(
71                      "Problema com o o arquivo de sistema do PDC");
72      }
73
74      @Test
75      public void testVerificaEstadoQuandoNaoLeModoOp()
76                                                      throws IOException {
77          when(bufReaderSysInfo.readLine()).thenReturn("[status] ligado");
78
79          assertEquals(null, verificador.verificaEstado());
80          verify(comunicador).guardarNoHistorico(
81                          "Problema com o o arquivo de sistema do PDC");
82      }
83
84      @Test
85      public void testVerificaEstadoExcecao() throws IOException {
86          when(bufReaderSysInfo.readLine()).thenThrow(new IOException());
87
88          assertEquals(null, verificador.verificaEstado());
89          verify(comunicador).guardarNoHistorico(
90                          "Problema com o o arquivo de sistema do PDC");
91      }
92
93      @Test
94      public void testObterAmostraNull() throws IOException {
95          when(bufReaderRegistroTemp.readLine()).thenReturn(null);
96
97          assertEquals(null, verificador.obterAmostras(2));
98          verify(comunicador).guardarNoHistorico(
99                                  "Problema no arquivo de amostras");
100     }
101
102     @Test
103     public void testObterAmostraNaoAtual() throws IOException {
104         when(bufReaderRegistroTemp.readLine()).thenReturn("[numAm] 3");
105
106         assertEquals(null, verificador.obterAmostras(2));
107         verify(comunicador).guardarNoHistorico(
```

```
108                                     "Arquivo de amostras desatualizado");
109         }
110
111         @Test
112         public void testObterAmostraNaoNumAm() throws IOException {
113             when(bufReaderRegistroTemp.readLine()).thenReturn(
114                                                     "[amostras] 7");
115
116             assertEquals(null, verificador.obterAmostras(2));
117             verify(comunicador).guardarNoHistorico(
118                                     "Problema no arquivo de amostras");
119         }
120
121         @Test
122         public void testObterAmostraSucesso() throws IOException {
123             when(bufReaderRegistroTemp.readLine()).thenReturn(
124                 "[numAm] 1 [dados] 18.8, -31.5, 29.8, 24.8,
125                 -13.8, 25.6, 13.7, -19.3, 17.8, 24.1, -17.4, 36.4 ");
126
127             float[] amostrasEsperadas = {18.8f, -31.5f, 29.8f, 24.8f,
128                 -13.8f, 25.6f, 13.7f, -19.3f, 17.8f, 24.1f, -17.4f, 36.4f};
129             float[] amostras = verificador.obterAmostras(1);
130             for (int i = 0; i < 12; i++) {
131                 assertEquals(amostrasEsperadas[i], amostras[i], 0.00001f);
132             }
133             verify(comunicador).guardarNoHistorico(
134                                         "Amostras obtidas com sucesso");
135         }
136
137         @Test
138         public void testObterAmostraExcecao() throws IOException {
139             when(bufReaderRegistroTemp.readLine()).thenThrow(
140                                                     new IOException());
141
142             assertEquals(null, verificador.obterAmostras(2));
143             verify(comunicador).guardarNoHistorico(
144                                     "Problema no arquivo de amostras");
145         }
146 }
```

- Testes da classe Comunidador.java

```
1 package suporte;
2
3 import java.io.BufferedWriter;
4 import java.io.IOException;
5 import org.junit.After;
6 import org.junit.AfterClass;
7 import org.junit.Before;
8 import org.junit.BeforeClass;
9 import org.junit.Test;
```

```
10 import static org.junit.Assert.*;
11 import static org.mockito.Mockito.*;
12
13 /**
14  *
15  * @author Camila Achutti
16  */
17 public class ComunicadorTest {
18
19     BufferedWriter instrucao;
20     BufferedWriter historico;
21     Comunicador comunicador;
22
23     public ComunicadorTest() {
24     }
25
26     @Before
27     public void setUp() {
28         instrucao = mock(BufferedWriter.class);
29         historico = mock(BufferedWriter.class);
30
31         comunicador = spy(new Comunicador(instrucao, historico));
32         when(comunicador.getDataTime()).thenReturn(
33                                         "dd/MM/yyyy HH:mm:ss.SSS");
34
35     }
36
37     @Test
38     public void testEmissaoDeComandoBemSucedida() throws IOException {
39
40
41         String comando = "example command";
42         comunicador.emitirComando(comando);
43
44         verify(instrucao).append(comando);
45         verify(instrucao).newLine();
46         verify(instrucao).flush();
47
48         String expectedToWriteOnHistorico =
49                 comunicador.getDataTime() + " " +
50                 comunicador.getDataTime() + " emitiu comando: " + comando;
51         verify(historico).append(expectedToWriteOnHistorico);
52         verify(historico).newLine();
53         verify(historico).flush();
54     }
55
56     @Test
57     public void testEmissaoDeComandoMalSucedida() throws IOException {
58         String comando = "example command";
59         comunicador.emitirComando(comando);
60     }
```

```
61
62      @Test
63      public void testGuardarHistoricoBemSucedida() throws IOException {
64          String evento = "relato evento";
65          comunicador.guardarNoHistorico(evento);
66          String expectedToWriteOnHistorico = comunicador.getDataTime()
67                                                          + " " + evento;
68          verify(historico).append(expectedToWriteOnHistorico);
69          verify(historico).newLine();
70          verify(historico).flush();
71
72      }
73
74      @Test
75      public void testGuardaHistoricoMalSucedida() throws IOException {
76          String evento = "evento";
77          comunicador.guardarNoHistorico(evento);
78          comunicador.emitirComando(evento);
79      }
80
81      @Test
82      public void testInstanciar() {
83          Comunicador result = Comunicador.instanciar();
84          assertTrue(result != null);
85      }
86  }
```

Apêndice D

Implementação das propriedades JPF produzidas

- P01 - IniciarCorrectlyProperty

```
1  public class IniciarCorrectlyProperty
2              extends PropertyListenerAdapter {
3    private int count = 0;
4
5    @Override
6    public boolean check(Search search, JVM vm) {
7      return (count > 0);
8    }
9
10   @Override
11   public void instructionExecuted(JVM vm) {
12     Instruction instruction = vm.getLastInstruction();
13     MethodInfo mi = instruction.getMethodInfo();
14     ThreadInfo ti = vm.getLastThreadInfo();
15     if (!ListenerUtils.filter(instruction)) {
16       System.out.println(">> "+ instruction.getSourceLine().trim());
17       System.out.println(mi.getFullName());
18       if (instruction.getSourceLine().trim().contains(
19             "this(Comunicador.instanciar(), Verificador.instanciar())")
20         count ++;
21     }
22   }
23
24   @Override
25   public String getErrorMessage() {
26     return "Property Violation: count = " + count;
27   }
28 }
```

- P02 - InstanciarVerificadorCorrectlyProperty

```
1  public class InstanciarVerificadorCorrectlyProperty
```

```
2                                          extends PropertyListenerAdapter {
3   private int count = 0;
4
5   @Override
6   public boolean check(Search search, JVM vm) {
7     return (count > 0);
8   }
9
10  @Override
11  public void instructionExecuted(JVM vm) {
12    Instruction instruction = vm.getLastInstruction();
13    MethodInfo mi = instruction.getMethodInfo();
14    ThreadInfo ti = vm.getLastThreadInfo();
15    if (!ListenerUtils.filter(instruction)) {
16      System.out.println(">> "+ instruction.getSourceLine().trim());
17      System.out.println(mi.getFullName());
18      if (instruction.getSourceLine().trim().contains("Verificador"))
19        count ++;
20    }
21  }
22
23  @Override
24  public String getErrorMessage() {
25    return "Property Violation: count = " + count;
26  }
27 }
```

- P03 - InstanciarComunicadorCorrectlyProperty

```
1  public class InstanciarComunicadorCorrectlyProperty
2                      extends PropertyListenerAdapter {
3   private int count = 0;
4
5   @Override
6   public boolean check(Search search, JVM vm) {
7     return (count > 0);
8   }
9
10  @Override
11  public void instructionExecuted(JVM vm) {
12    Instruction instruction = vm.getLastInstruction();
13    MethodInfo mi = instruction.getMethodInfo();
14    ThreadInfo ti = vm.getLastThreadInfo();
15    if (!ListenerUtils.filter(instruction)) {
16      if (instruction.getSourceLine().trim().contains("Comunicador") &&
17                      mi.getFullName().contains("Comunicador"))
18        count ++;
19    }
20  }
21
22  @Override
```

```
23    public String getErrorMessage() {
24      return "Property Violation: count = " + count;
25    }
26  }
```

- P04 - ConstructedBufferSimplesCorrectlyProperty

```
1   public class ConstructedBufferSimplesCorrectlyProperty
2                       extends PropertyListenerAdapter {
3     private int count = 0;
4
5     @Override
6     public boolean check(Search search, JVM vm) {
7       return (count > 0);
8     }
9
10    @Override
11    public void instructionExecuted(JVM vm) {
12      Instruction instruction = vm.getLastInstruction();
13      MethodInfo mi = instruction.getMethodInfo();
14      ThreadInfo ti = vm.getLastThreadInfo();
15      if (!ListenerUtils.filter(instruction)) {
16        if (mi.getFullName().contains("alocar"))
17          count ++;
18      }
19    }
20
21    @Override
22    public String getErrorMessage() {
23      return "Property Violation: count = " + count;
24    }
25  }
```

- P05 - AllocatedCorrectlyProperty

```
1   public class AllocatedCorrectlyProperty
2                       extends PropertyListenerAdapter {
3     private int count = 0;
4
5     @Override
6     public boolean check(Search search, JVM vm) {
7       return (count > 0);
8     }
9
10    @Override
11    public void instructionExecuted(JVM vm) {
12      Instruction instruction = vm.getLastInstruction();
13      MethodInfo mi = instruction.getMethodInfo();
14      ThreadInfo ti = vm.getLastThreadInfo();
15      if (!ListenerUtils.filter(instruction)) {
16        if (instruction.getSourceLine().trim().contains(
```

```
17                                  "ByteBuffer.allocateDirect"))
18          count ++;
19        }
20      }
21
22      @Override
23      public String getErrorMessage() {
24        return "Property Violation: count = " + count;
25      }
26    }
```

- P06 - InserirTemperaturaCorrectlyProperty

```
1   public class InserirTemperaturaCorrectlyProperty
2                       extends PropertyListenerAdapter {
3     private int count = 0;
4
5     @Override
6     public boolean check(Search search, JVM vm) {
7       return (count > 0);
8     }
9
10    @Override
11    public void instructionExecuted(JVM vm) {
12      Instruction instruction = vm.getLastInstruction();
13      MethodInfo mi = instruction.getMethodInfo();
14      ThreadInfo ti = vm.getLastThreadInfo();
15
16      if (!ListenerUtils.filter(instruction)) {
17        System.out.println(">> "+ instruction.getSourceLine().trim());
18        System.out.println(mi.getFullName());
19        if (instruction.getSourceLine().trim().contains("putFloat"))
20          count ++;
21      }
22    }
23
24    @Override
25    public String getErrorMessage() {
26      return "Property Violation: count = " + count;
27    }
28  }
```

- P07 - RecuperarTemperaturaCorrectlyProperty

```
1   public class RecuperarTemperaturaCorrectlyProperty
2                         extends PropertyListenerAdapter {
3     private int count = 0;
4
5     @Override
6     public boolean check(Search search, JVM vm) {
7       return (count > 0);
```

```
8     }
9
10    @Override
11    public void instructionExecuted(JVM vm) {
12      Instruction instruction = vm.getLastInstruction();
13      MethodInfo mi = instruction.getMethodInfo();
14      ThreadInfo ti = vm.getLastThreadInfo();
15      if (!ListenerUtils.filter(instruction)) {
16        System.out.println(">> " + instruction.getSourceLine().trim());
17        System.out.println(mi.getFullName());
18        if (instruction.getSourceLine().trim().contains("getFloat"))
19          count ++;
20      }
21    }
22
23    @Override
24    public String getErrorMessage() {
25      return "Property Violation: count = " + count;
26    }
27
28  }
```

Apêndice E

Mains alternativos produzidos para a verificação

- Main Para Verificação do Componente de Dados

```
1  package swpdc;
2  import dados.BufferSimples;
3
4  public class MainParaVerificacaoDados {
5      private static BufferSimples m_bufTta;
6
7      public static void main(String[] args){
8          float m_buffer[] = {1,3,4,5,6,7,8,9,4,5,6,7};
9          Object area = new Object();
10         m_bufTta = new BufferSimples(area , (byte)60000);
11         m_bufTta.inserir(m_buffer[1], (byte)4, 1);
12         float result = m_bufTta.recuperar(0);
13         System.out.println(result);
14     }
15 }
```

- Main Para Verificação do Componente de Suporte

```
1  package swpdc;
2  import suporte.Comunicador;
3  import suporte.Iniciador;
4  import suporte.Verificador;
5
6  public class MainParaVerificacaoSuporte {
7      static Comunicador comunicador;
8      static Verificador verificador;
9      static Iniciador iniciador;
10
11     public static void main(String[] args){
12         iniciador = new Iniciador();
13         verificador = Verificador.instanciar();
14         comunicador = Comunicador.instanciar();
```

```
15            iniciador.iniciar();
16        }
17    }
```