Utilização de mineração de especificação na identificação de fluxos inválidos em softwares

Luciano Kelvin da Silva

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientadora: Profa. Dra. Ana Cristina Vieira de Melo

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, Junho de 2016

Utilização de mineração de especificação na identificação de fluxos inválidos em softwares

Esta é a versão original da dissertação elaborada pelo
candidato Luciano Kelvin da Silva, tal como
submetida à Comissão Julgadora.

Utilização de mineração de especificação na identificação de fluxos inválidos em softwares

Esta versão da dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 15/06/2016. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Profª. Drª. Ana Cristina Vieira de Melo (orientadora) - IME-USP
- Profª. Drª. Kelly Rosa Braghetto - IME-USP
- Prof. Dr. Nandamudi Lankalapalli Vijaykumar - INPE

Agradecimentos

Agradeço a Deus por ter me ajudado e capacitado durante toda a realização deste trabalho.

Gostaria de agradecer a Dr. Ana Cristina Vieira de Melo pela paciência e todo conhecimento transmitido durante esse período.

Agradeço a minha mãe Maria do Socorro da Silva e minha irmã Luciana Maria Silva Ferreira por terem me apoiado desde o início de minha vida acadêmica e por terem me dado o suporte necessário para que eu chegasse até aqui.

Agradeço a minha namorada Renata Maria Meireles Silva por todo o carinho, palavras de apoio, e pelo companheirismo demonstrado durante todos os dias da realização deste mestrado.

Agradeço ao amigo Alexandre Locci por toda a ajuda e opiniões dadas, as quais só contribuíram para o desenvolvimento deste trabalho.

Aos amigos que fiz no IME, em especial aos amigos do LIAMF, pela amizade e companheirismo dos últimos dois anos.

Por fim, agradeço aos irmãos Moisés Altino e Marli Borges, assim como toda sua família, por terem me acolhido e se tornado minha família nesses últimos dois anos, e que continuarão sendo mesmo após o fim desse mestrado.

Resumo

SILVA, L. K. **Utilização de mineração de especificação na identificação de fluxos inválidos em softwares**. 2016. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

A presença dos softwares em nosso cotidiano é cada vez maior. Eles são responsáveis por tarefas que vão do envio de um e-mail até o controle de sistemas complexos como usinas nucleares e aeronaves. Devido a essa presença constante e crescente, é necessário que estes softwares apresentem o menor número de falhas possível a fim de evitar situações, potencialmente catastróficas, que podem resultar em perdas de bens e/ou vidas. A fim de identificar a presença de falhas de software antes que tais sistemas sejam colocados em produção, são utilizadas duas metodologias conhecidas como Testes de Software e Verificação Formal. Apesar de muito distintas, estas duas metodologias possuem em comum a necessidade de realizar uma interpretação das especificações do software visando produzir casos de teste (Testes de Software) ou modelos formais (Verificação Formal). Este processo de interpretação é manual e, via de regra, muito custoso em termos de homens/horas. Motivados pela identificação desta limitação, comum às duas técnicas, e pela possibilidade de redução de custos via automação deste processo, propomos uma metodologia que produz, via mineração de especificação do software implementado em Java, um modelo que represente o comportamento real do software, no formato de uma Máquina de Estados Finitos e compara de forma automática este modelo com a especificação descrita em *Statechart*. Tal comparação permite verificar se o software está de acordo com as definições feitas no projeto da aplicação. Pretendemos com esta metodologia dar suporte aos desenvolvedores para que estes identifiquem possíveis fluxos de execução que estão presentes no código-fonte, mas que não são permitidos pelas especificações. A existência de um fluxo na implementação que não esteja descrito na especificação pode ser um forte indicador de que existe um erro na implementação.

Palavras-chave: Mineração de Especificação, *Statecharts*, Java, Máquina de Estados Finitos.

Abstract

SILVA, L. K. **Specification mining use in identifying invalid flows in softwares**. 2016. Dissertation (Master's Degree) - Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2016.

The presence of software-controlled systems is growing in our daily lives. They are responsible for many different tasks such as sending an email, managing a nuclear facility or flying an aircraft. For this reason, it is necessary ensuring a high software quality in order to promote a real reduction of the faults to avoid situations that could lead to loss of properties as well as lives. To identify the presence of software errors that could generate fault before the systems become fully operational, the software industry relies on two main methods to identify software erros: software testing and formal verification. Although very different, both methods have the common necessity of the interpretation of the software specifications in order to produce test cases (software testing) or formal models (formal verification). In most cases, this process is performed manually what increases the cost of software verification. Driven by the identification of this drawback in both techniques, we propose a methodology that produces via specification mining of softwares written in Java, a model that represents the actual behavior of the software as a Finite State Machine and automatically compares this model with the specification described in Statechart. This comparison allows to check if the software is in accordance with the rules made in the specification of the project. We intend with this methodology to support developers such that it could be possible to automatically indentify execution traces that are present in the source code but that are not allowed by the specifications. The existence of a trace in the implementation that is not described in the specification can be a strong indicator that there is an error in the implementation.

Keywords: Specification Mining, Statecharts, Java, Finite State Machine.

Sumário

Lista de Abreviaturas

GFC	Grafo de Fluxo de Controle
PPF	Possível Ponto de Falha
SM	*Software Mining*
UML	*Unified Modeling Language*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A presença dos softwares em nosso cotidiano é cada vez maior. Eles são responsáveis por tarefas que vão do envio de um e-mail até o controle de complexos sistemas como usinas nucleares e aeronaves. Devido a essa presença constante e crescente, é necessário que estes softwares apresentem o menor número de falhas possíveis a fim de evitar situações, potencialmente catastróficas, que podem resultar em perdas de bens e/ou vidas

Uma **falha** é qualquer comportamento que não está definido nas especificações do software. Sendo assim, é um comportamento inesperado e/ou indesejável manifestado pelo software em tempo de execução que não está em conformidade com o comportamento esperado determinado em projeto. Por sua vez, este tipo de comportamento indesejável e/ou inesperado ocorre devido à introdução de erros no software durante as fases de desenvolvimento da aplicação. Um **erro** é o resultado da ação humana incorreta ou imprecisa motivada pelo cansaço, desatenção, desconhecimento técnico e/ou etc. [AO08]. Logo, identificar erros é uma atividade importante no processo de desenvolvimento de um software, pois evita a ocorrência de falhas que podem gerar consequências graves e dispendiosas.

Apesar de sua importância, o processo de identificação de erros é reconhecidamente oneroso, sendo o seu custo sensivelmente menor quando os erros são identificados ainda nas fases de desenvolvimento nas quais foram inseridos. Ou seja, um erro identificado em uma etapa de teste unitário terá um custo de correção menor do que se este mesmo erro fosse localizado em fases posteriores de desenvolvimento como, por exemplo, integração ou aceitação. Estudos apontam que identificar um erro que gerou uma falha com o software já em produção pode ser até 500 vezes mais custoso do que identificar este mesmo erro durante o processo de desenvolvimento [BK+08].

A fim de localizar erros durante o processo de desenvolvimento de uma aplicação, a indústria de software dispõe de várias metodologias, sendo as duas mais comuns e utilizadas conhecidas como **Teste de Software** [AO08] e **Verificação Formal** [BK+08]. O teste de software utiliza a observação do comportamento estimulado do software, através de um subconjunto de valores pertencentes ao conjunto de entradas ("inputs") do software que se deseja testar. O comportamento observado é então confrontado com o comportamento esperado (definido nas especificações do software). Se ocorrer discrepância entre os comportamentos observados e o esperado, então é admitido que existe um erro no software. Por sua vez, a metodologia de verificação formal baseia-se na representação de um software sob verificação na forma de modelo ou conjunto de expressões lógicas. Para verificação automatizada, o software é representado na forma de um modelo, e as propriedades desejadas são definidas em uma linguagem de modelagem. Posteriormente, o modelo é verificado por uma ferramenta denominada *Model Checker*, que tenta violar a propriedade especificada. Se o *model checker* consegue violar a propriedade, então é aceito que um erro foi encontrado, caso contrário, é aceito que o software pode violar a propriedade, o que implica que está correto. No caso da representação

ser feita na forma de expressões lógicas, é realizada uma prova de teorema a fim de encontrar inconsistências nas expressões. Se nada for encontrado, é aceito que o software está livre de erros.

Embora sejam metodologias distintas, ambas possuem um fator em comum que consiste na interpretação das especificações do software, feita pelos desenvolvedores, a fim de construir as malhas de testes (teste de software) ou os modelos formais com as respectivas propriedades a serem verificadas (verificação formal). Isto implica em uma atividade manual que possui as seguintes limitações:

- está vinculada às capacidades individuais de cada desenvolvedor;
- pode consumir muitos recursos em termos de homens/hora; e
- está sujeita a erros de julgamento por parte dos desenvolvedores no momento da escolha dos casos de teste ou das propriedades a serem verificadas.

Sensibilizados por estes limites e pela consciência de que a identificação de erros durante o processo de desenvolvimento é de vital importância tanto para redução de custos de projeto como na prevenção de falhas que podem ter consequências catastróficas, propomos uma metodologia para auxiliar a verificação da corretude do software utilizando Mineração de Especificação [LKHL11] e comparação de sistema e especificação. Nesta metodologia extraímos uma **Máquina de Estados Finitos** que representa o comportamento de uma parte do software, via mineração de especificação a partir do código-fonte, e a comparamos com a especificação inicial definida em ***Statechart*** [Har87], buscando saber se todo comportamento (*trace*) possível no modelo da implementação está previsto na especificação, sendo assim uma correta implementação da especificação. Na Figura 1.1, temos uma visão geral da abordagem proposta.

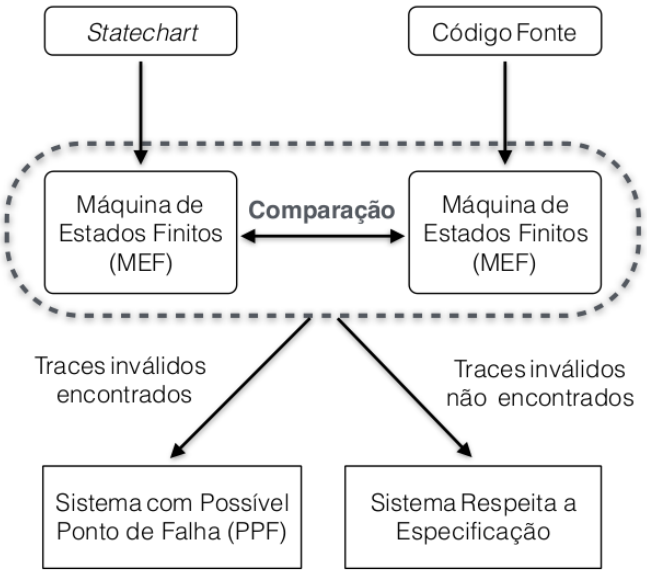


Figura 1.1: *Fluxo da abordagem.*

A mineração de especificação é uma técnica baseada em mineração de dados que possibilita obter diversos tipos de informações e representações de um software, entre elas podemos destacar [LKHL11]:

- máquinas de estados;
- invariantes;

- padrões de comportamentos; e
- diagramas de sequências

tudo a partir da análise do código-fonte ou da observação do software em funcionamento.

A mineração que observa o software em funcionamento, **Análise Dinâmica**, visa monitorar como o software se comporta em determinadas situações, extraindo informações sobre os caminhos de chamadas de métodos e valorações de variáveis. Entretanto essa técnica fica dependente dos cenários executados durante a observação, e com isso ela pode não representar todas as possíveis execuções do software. Em contra-partida, a mineração de especificação que analisa o código-fonte, **Análise Estática**, não necessita da execução do programa, pois a análise é feita sobre as estruturas contidas na codificação do software. Esta abordagem permite obter uma visão mais completa sobre os possíveis *traces* do sistema por não depender de um cenário de execução. Porém essa técnica não possibilita analisar quais valores podem ser, ou não, atribuídos a variáveis ou objetos [LKHL11]. Logo, cada uma dessas técnicas possui vantagens e desvantagens, podendo inclusive serem usadas em conjunto com o objetivo de produzir um modelo mais rico em informação.

Um dos modelos que pode ser gerado por mineração de especificação, e um dos mais utilizados, é um grafo que representa os fluxos de chamadas de métodos do software (e.g. [MSY12], [PSYY13], [CBGU13]). Através desse grafo é possível entender o funcionamento dos *traces* do sistema, ou seja, os possíveis caminhos que o software pode percorrer em determinadas situações. As máquinas de estados finitos são um dos modelos de grafos mais simples de se manipular e entender, o que a torna uma representação adequada para compreender o fluxo de processos de um sistema, além de todo o corpo teórico já desenvolvido para verificar a qualidade de software.

Por fim, propomos uma metodologia semi-automática para verificação da corretude do software mediante uma comparação entre um modelo da especificação e a sua implementação (código-fonte). Para esta comparação utilizaremos **Mineração de Especificação** [LKHL11] para obter uma **Máquina de Estados Finitos** que represente os possíveis comportamentos do software, e utilizaremos o modelo ***Statechart*** [Har87] como modelo da especificação. A partir dessa comparação, esperamos que seja possível identificar fluxos inválidos que tenham sido implementados no software, mas não são previstos pela especificação de projeto, sendo esses **Possíveis Pontos de Falhas** (PPFs), assim como verificar se a especificação está realmente cobrindo todos os fluxos que poderão ocorrer quando o software for executado. Esta metodologia será aplicada na linguagem de programação Java, visto que se trata de uma linguagem bem difundida, com uma sintaxe simples e semelhante a de outras linguagens bem conhecidas como *C#* e *C++*. Na Figura 1.2 podemos ver um fluxograma de como funciona todo o processo da abordagem que propomos, nela é possível ver o processo de transformação pelo qual os modelos passarão até o momento da comparação. O fluxograma é composto pelos seguintes componentes:

- Elipses com borda contínua: Softwares e *frameworks* utilizados;
- Elipses com borda tracejada: Algoritmos que implementamos;
- Quadrados com cantos arredondados: Modelos utilizados neste trabalho;
- Quadrados com cantos pontiagudos: Resultados obtidos ao final da abordagem;
- Setas: Formato em que o modelo se encontra.

1.2 Objetivos

Os objetivos deste trabalho são:

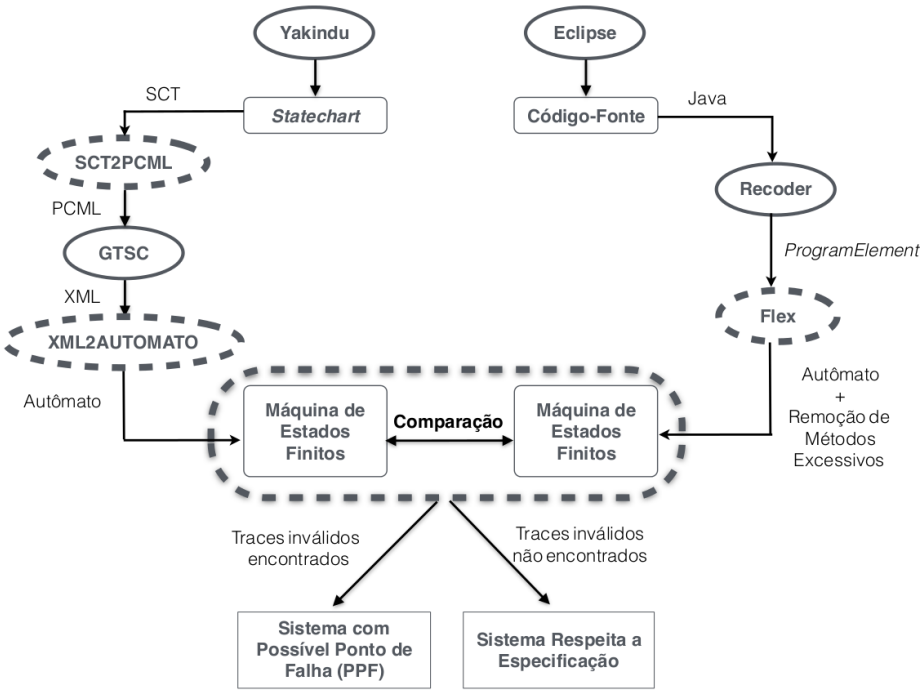


Figura 1.2: *Fluxo detalhado da abordagem.*

- Colaborar com pesquisas nas áreas de teste de software e verificação formal;
- Auxiliar desenvolvedores na busca por falhas em softwares;
- Auxiliar a identificação de problemas nas especificações dos softwares;
- Apresentar uma metodologia que permita aos desenvolvedores identificar discordâncias entre os possíveis fluxos de chamadas de métodos que possam ocorrer no software em relação a sua especificação em *statechart*. Para isso será necessário:
  - Implementar um algoritmo de análise estática que permita criar uma máquina de estados finitos com as possíveis sequências de chamadas de métodos em um determinado cenário no sistema;
  - Utilizar um algoritmo que permita transformar um *statechart* em uma máquina de estados finitos;
  - Implementar um algoritmo que possibilite identificar as diferenças entre duas máquina de estados finitos;
- Aplicar essa metodologia em sistemas reais, e avaliar os benefícios e limitações.

1.3 Trabalhos Relacionados

A mineração de especificação é uma metodologia que permite extrair informações sobre um dado software. Para tanto, ela utiliza análises estáticas e dinâmicas sobre o software, as quais produzem dados que são de enorme utilidade para manutenção, verificação e validação do próprio software. A fim de realizar essas análises, existem duas metodologias dentro de mineração de especificação que são bastante utilizadas [KBP+10]:

- inferência de invariantes, e
- inferência de modelos baseados em estados.

A inferência de invariantes permite obter quais restrições o sistema pode ter em determinados momentos ou uma restrição geral para o sistema. Por sua vez, a inferência de modelos baseados em estados permite a construção de uma "visão geral" dos possíveis *traces* do software. Estes *traces* podem ser obtidos tanto observando o software em funcionamento, análise dinâmica, como através da análise das estruturas de controle no código-fonte, análise estática [PSYY13, SYFP08, LGW09, WML02, JS08, FYD+08].

Nas seções que seguem, veremos alguns trabalhos que buscam obter modelos de transição a partir do software, com diferentes tipos de abordagens e diferentes objetivos para a utilização desses modelos.

1.3.1 Mineração de *Traces* do Sistema

Peleg et al. [PSYY13] apresenta uma análise estática baseada na observação dos usos de uma classe a fim de montar a especificação da mesma. Para tanto, são observados os possíveis fluxos de métodos de todos os objetos que sejam instâncias dessa classe. Posteriormente, para cada uma das instâncias é montada uma máquina de estados finitos com os *traces* minerados. Por fim, essas máquinas de estados finitos são resumidas, mescladas e agrupadas em função das frequências de ocorrência dos eventos observados. Uma vez produzida a modelagem do código na forma de uma máquina de estados finitos, a metodologia de Peleg realiza sucessivas depurações do modelo por meio da mescla ou remoção de fluxos. Em nosso trabalho, usamos o processo de construção de uma máquina de estados finitos a partir da observação do software sob verificação, contudo, nós realizamos um processo de depuração diferente em nossas máquinas de estados, pois estamos interessados na identificação de todos os possíveis fluxos, válidos ou inválidos, que podem ser identificados no software, e para isso necessitamos do máximo de informação disponível para a comparação com a especificação.

Krka et al. [KBP+10] minera um autômato que representa o fluxo de chamadas de métodos permitidos no software, além de invariantes do software, a fim de combiná-los. O objetivo dessa associação é reduzir as perdas de informações ao mesclar e podar o autômato extraído. Inicialmente são obtidas os invariantes dos objetos do software, e depois realizam uma análise dinâmica, que observa os *traces* do sistema em funcionamento, para construir uma máquina de estados finitos desses *traces*. Por fim, eles utilizam os invariantes como meio para escolher quais *traces* podem ser removidos da máquina de estados finitos que foi extraída, e verificar se algum *trace* viola os invariantes. Por se tratar de uma análise dinâmica, é possível que alguns *traces* não sejam observados pelo simples fato de não terem sido executados durante o tempo de realização da verificação. Nosso trabalho diferencia-se do trabalho de Krka et al. [KBP+10] por utilizar uma análise estática a qual permite identificar todos os possíveis fluxos, válidos ou inválidos, mesmo que estes não venham a ser executados. Além disso, não utilizamos invariantes, já que não pretendemos resumir os modelos extraídos, e a verificação de invariantes está fora do escopo do nosso trabalho.

Os trabalhos de Peleg et al. [PSYY13] e Krka et al. [KBP+10] buscam minerar modelos que tornem a especificação mais precisa de modo que esta expresse o que realmente se encontra codificado no software. Por esse motivo, otimizam os modelos para que esses fiquem mais exatos a fim de facilitar a interpretação por parte dos desenvolvedores. Porém, autômatos minerados a partir do software também podem ser utilizados para identificar *traces* inválidos presentes no software. A fim de identificar *traces* inválidos em um software, Wasylkowski et al. [WZL07, Was10] propõem uma técnica que identifica os fluxos de execução que possivelmente estão errados no software. A identificação destes fluxos é feita por meio da construção de uma máquina de estados finitos com o comportamento de cada um dos objetos de uma classe pré-definida, o que permite reconhecer quais padrões de fluxos existem entre esses objetos. Posteriormente, é verificado quais objetos possuem *traces* que não respeitam esse padrão. *Traces* que estão fora do padrão são identificados como anô-

malos e podem ser um forte indicativo da existência de erro no software.

De maneira similar, os trabalhos de Caso et al. [dCBGU11, CBGU13] constroem uma máquina de estados por meio da mineração de especificação estática, o qual representa o comportamento de objetos específicos no software; Após isso são definidos invariantes para cada estado, sendo que um estado é o momento do objeto após a execução de uma ação, e um invariante é uma restrição dada a esse estado, como por exemplo, a proibição da chamada de algum método em determinado momento. Se em algum momento um fluxo viola uma desses invariantes, é considerado que um possível erro no software foi identificado.

1.3.2 Verificando o Sistema com a Especificação

Como já vimos, modelos minerados a partir do software já vêm sendo utilizados na identificação de *bugs*, através de técnicas como identificação de padrões de uso, violação de restrições e comparações com modelos da especificação. A técnica que busca realizar uma comparação com os modelos da especificação é uma abordagem bastante interessante, visto que geralmente os modelos da especificação já existem na documentação do software, além de possuírem regras bem definidas do comportamento desejado do sistema. A seguir vemos alguns trabalhos que buscam traçar esse paralelo entre um modelo da especificação e o software já implementado na busca de possíveis falhas no sistema.

Drusinsky et al. [DSD05] propõem uma técnica baseada no monitoramento do software durante sua execução (*Run-time Execution Monitoring*), no qual é verificado se um *trace* viola algum dos invariantes definidos em um modelo gerado da especificação do software. Desta forma, se um *trace* viola um invariante, é aceito que um *trace* inválido foi encontrado. Diferentemente do trabalho de Drusinsky, nosso trabalho utiliza modelos da especificação do software já modelada na forma de diagramas *statecharts*, ao invés de definir invariantes. Além disso, não utilizamos modelos baseados em invariantes, e sim buscamos identificar diferenças entre os modelos gerados a partir dos diagramas *statecharts* e o modelo gerado a partir do código-fonte da aplicação. Essas diferenças são para nós indicativos de *traces* inválidos.

Fink et al. [FYD+08] nos apresenta a uma técnica que verifica se um programa está de acordo com um *typestate* [SY86] dado como especificação. Nessa abordagem, vemos a obtenção de uma máquina de estados finitos com os fluxos dos objetos que pertencem à classe definida na especificação, e posteriormente são utilizados alguns verificadores para checar as propriedades do *typestate* nos fluxos do objeto, sempre calculando a medida tempo/precisão de cada um desses verificadores, buscando ver o que possui melhor custo/benefício. Essa abordagem busca identificar quais caminhos conduzem o programa para um estado de erro, estado esse que é definido no *typestate* da especificação. Em nossa abordagem utilizamos modelos *statecharts* como modelos da especificação do cenário do software, sabendo que esse tipo de especificação já é bastante usada, e possui uma boa capacidade para descrição para fluxos de processos.

O trabalho de Kumar et al. [KKN15] nos mostra uma análise estática sobre programas de processamento de arquivos. Nesta abordagem é minerada uma máquina de estados finitos que representa os possíveis *traces* de operações que estão sendo realizadas sobre os arquivos manipulados, e depois compara este autômato com um modelo denominado "*format-file*" que representa a especificação do arquivo no formato de autômato. Através dessa comparação, é possível identificar se os programas têm "*overacepted*", ou seja, está aceitando manipulações no arquivo a mais do que está definido na especificação, denominada "*format-file*". A comparação permite também saber se o programa está "*underacepted*", ou seja, não aceita uma sequências de comandos definidos na especificação. O trabalho de Kumar se assemelha ao nosso, visto que extrai um autômato de maneira estática, e o utiliza para comparar com uma especificação buscando realizar uma verificação no software. Contudo ele

se deteve em um foco bem específico, manipulação de arquivos, e também não utiliza recursão em seus algoritmos, inviabilizando o uso de estruturas de controle aninhadas no código-fonte. Em nosso trabalho permitimos observar qualquer tipo de sistema, não somente de manipulação de arquivos, desde que este software esteja na linguagem Java e tenha sua especificação em *statechart*, além de trabalharmos com um algoritmo recursivo, ou seja, não há problemas com estruturas aninhadas.

Foster et al. [FUMK03] tem como objetivo descobrir erros de fluxos em *webservices*. Para isso eles partem do princípio de que já existe uma especificação dos fluxos corretos do software no formato de um FSP (*Finite State Process*), depois são modelados os fluxos reais do sistema no formato BPEL4WS (*Business Process Language for Web Services*), um tipo de notação baseada em XML. Para identificar os possíveis fluxos inválidos, primeiramente o modelo BPEL4WS é convertido para um FSP, e depois comparado com o FSP da especificação, buscando com isso os possíveis erros que serão os fluxos que divergem entre os dois modelos. Este trabalho possui uma grande interseção com o nosso, visto que compara uma especificação baseada em estados com o comportamento real do software. Contudo nós mineramos o comportamento do software a partir do seu código-fonte, ao passo que o trabalho de Foster constrói o modelo BPEL4WS (que no decorrer do processo é transformado em um FSP), além do trabalho de Foster ser focado no contexto de serviços web, enquanto o nosso foca em sistemas orientados a objetos.

Dentre os trabalhos vistos, percebemos que alguns utilizam as máquinas de estados finitos para identificar possíveis erros no fluxo do software [WZL07, Was10, dCBGU11, CBGU13, FYD+08, KKN15, FUMK03], entretanto nenhum tentou fazer uma relação de ordem parcial entre estes modelos extraídos do sistema com modelos *statecharts* [Har87] definidos na especificação do software, e através dessa relação, apontar se a máquina de estados finitos do sistema é uma implementação correta da especificação.

1.4 Organização do Trabalho

O restante deste trabalho está dividido da seguinte maneira:

- **Capítulo 2:** apresenta os conceitos sobre *statechart*, que será o modelo utilizado como especificação neste trabalho, e também mostra como este modelo será transformado em uma máquina de estados finitos, afim de que possa ser utilizado na comparação com a implementação;
- **Capítulo 3:** detalha como é obtida uma máquina de estados finitos que represente o comportamento de um objeto dentro do software, mostrando os conceitos e algoritmos utilizados;
- **Capítulo 4:** explica os conceitos e algoritmos utilizados para verificar a relação de conformidade entre os modelos da implementação e especificação;
- **Capítulo 5:** apresenta estudos de caso feitos nos quais testamos nossa abordagem em softwares de terceiros escritos na linguagem Java, e com especificações descritas em *statechart*;
- **Capítulo 6:** Discute os resultados obtidos, indicações de trabalhos futuros e conclusões.

Capítulo 2

Modelo da Especificação

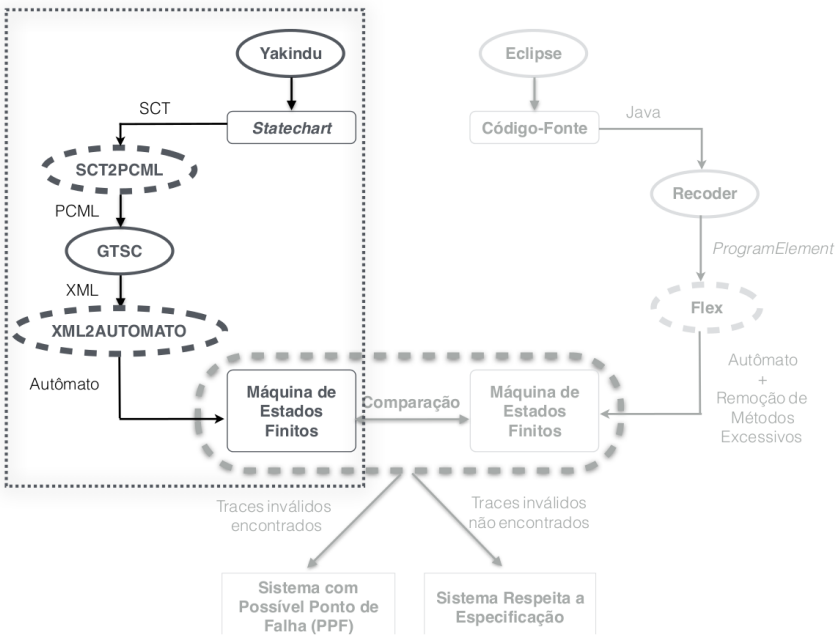


Figura 2.1: *Fluxo completo da abordagem (Especificação).*

Neste trabalho é utilizado o diagrama de *statechart* como modelo de especificação para descrever o comportamento que pode ser executado por um objeto dentro do software. Para tornar possível a realização da comparação entre a especificação e o modelo que representa a implementação do sistema, uma máquina de estados finitos, será necessário transformar esse *statechart* em uma máquina de estados finitos. Este capítulo dará uma visão geral sobre as máquinas de estados finitos e os *statecharts*, além de apresentar as transformações aplicadas ao modelo *statechart* da especificação para que este seja transformado em uma máquina de estados finitos.

2.1 Máquinas de Estados Finitos

As máquinas de estados finitos são grafos direcionados, que permitem representar comportamentos e montar sequências de ações. São formadas por um conjunto finito de estados e um conjunto finito de transições, sendo que cada transição liga um estado a outro, ou a ele mesmo. Possuem diversos usos como: testar circuitos, modelar o fluxo de sistemas computacionais, entre outros.

As máquinas de estados podem ser divididas em dois grupos, sendo eles:

- Máquinas tradutoras ou autômatos finitos com saída, as quais há uma saída referente para cada entrada na máquina;
- Máquinas reconhecedoras ou autômatos finitos, as quais cada entrada tem duas possibilidades de saída, sendo elas **aceita** ou **rejeita**, representando a aceitação ou não da palavra de entrada na linguagem da máquina de estados.

Neste trabalho iremos modelar os fluxos do software através de máquinas de estados reconhecedoras, visto que o modelo que precisamos para representar o fluxo do sistema, é uma máquina de estados finitos que reconheça todos os possíveis fluxos que possam ocorrer com um objeto em uma parte do software. Sabendo disso usaremos uma adaptação da definição de máquinas de estados finitos [CBM89] onde removemos o alfabeto de saída e a função de saída, visto que não necessitamos dessas propriedades para nosso modelo. Portanto a definição de máquina de estados finitos utilizada neste trabalho é:

Definição 1 *Máquina de estados finitos é uma tupla composta por quatro elementos,* $(Q, \Sigma, \delta, q0)$.

- Q*: Conjunto não vazio de estados;*
- Σ*: É um conjunto finito chamado alfabeto de entrada;*
- δ*:* $Q \times \Sigma \rightarrow Q$ *é a função de transição de estados;*
- $q0 \in Q$ *é o estado inicial;*

Graficamente, as máquinas de estados são simples de representar. Nelas, cada estado é retratado como um círculo e cada transição é representada por uma seta ligando um estado A a um estado B (ou a ele mesmo), rotulada com um símbolo α do alfabeto, para $\alpha \in \Sigma$, e o estado inicial é um estado apontado por uma seta sem rótulo e sem um estado de origem.

Para exemplificar melhor as máquinas de estados, temos na Figura 2.2 o modelo de uma máquina de estados representando a especificação de uma classe `Socket` com as seguintes restrições:

- Qualquer operação de `getInputStream` ou `getOutputStream` só pode ser feita após a chamada do método `connect`;
- Sempre ao final deve ser utilizado o método `close` para finalizar as operações.

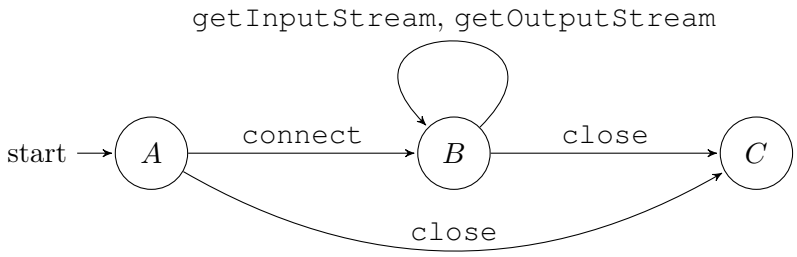


Figura 2.2: *Máquina de Estados referente a especificação da classe Socket.*

A configuração da máquina de estados da Figura 2.2, é a seguinte:

- $Q = \{A, B, C\}$;
- $\Sigma = \{$`connect, getInputStream, getOutputStream, close`$\}$;
- $q0 = A$;
- δ: A função de transição é descrita na Tabela 2.1.

	A	B	C
connect	$\{B\}$	$\{\}$	$\{\}$
getInputStream	$\{\}$	$\{B\}$	$\{\}$
getOutputStream	$\{\}$	$\{B\}$	$\{\}$
close	$\{C\}$	$\{C\}$	$\{\}$

Tabela 2.1: *Função de transição da máquina de estados finitos da Figura 2.2.*

2.1.1 Limitações

O crescimento do tamanho e da complexidade dos sistemas acaba ocasionando uma explosão de estados durante a construção das máquinas de estados, o que dificulta seu entendimento. Outra limitação das máquinas de estados finitos é a dificuldade em descrever ações paralelas e concorrentes, que são recursos bastante utilizados nos sistemas computacionais atuais. Diante desses problemas, a literatura aponta alguns modelos que dispõem de características para superar estas limitações, dentre eles os mais populares segundo Crane [CD05] são: os *statecharts* propostos por Harel [HPSS87], os diagramas de estados que fazem parte da UML [BRJ97], e o Rhapsody [HK04], sendo este uma versão orientada a objetos dos *statecharts* definido por Harel. Neste trabalho optou-se por abordar os *statecharts* clássicos definidos por Harel como modelo formal da especificação do sistema. Na próxima seção, será apresentada uma visão geral sobre os *statecharts*, assim como seus componentes e sua representação gráfica.

2.2 *Statecharts*

Com o desenvolvimento dos Sistemas Reativos [MP92][HP85], percebeu-se a dificuldade de descrever o comportamento desse tipo de software de maneira mais clara, objetiva, e ainda assim, precisa e formal. Para solucionar esse problema Harel propôs os ***Statecharts*** [Har87] como uma extensão das máquina de estados, com novas funcionalidades como: ortogonalidade (ações paralelas), hierarquia de estados e comunicação em *broadcast*. O *statechart* pode ser definido como um formalismo visual usado para descrever o comportamento de sistemas reativos e de tempo-real.

Os *statecharts* foram adotados como um padrão da UML [BRJ97] e são usados para descrever o comportamento do sistema. Na versão 2.0 da UML, os *statecharts* sofreram algumas modificações e passaram a ser chamados de Diagrama de Estados. Neste trabalho iremos utilizar o termo *statechart* para nos referir ao padrão definido por Harel [Har87].

2.2.1 Componentes

A sintaxe dos *statecharts* foi descrita por Harel detalhadamente em [HPSS87], onde é explicado como manuseá-los. A seguir daremos uma visão geral sobre os componentes que serão abordados neste trabalho.

- **Estados**: representam uma situação em que o sistema pode se encontrar em um determinado momento. Dentro do conjunto de estados, existe o subconjunto de estados iniciais, sendo eles estados padrão de um *statechart* antes de qualquer processamento;
- **Transições**: representam ações do sistema que conduzem de um estado para outro(s);
- **Hierarquia ou decomposição *XOR* (ou exclusivo)**: permite organizar e reduzir a quantidade de estados e transições. Esse recurso permite criar um estado pai, chamado de estado-OR,

onde se pode colocar sub-rotinas, ou agrupar estados com características semelhantes. Também permite dividir o *statechart* em vários documentos, onde um subnível de um estado pode ser representado em um documento diferente, permitindo uma maior organização;

- **Ortogonalidade ou decomposição *AND* de estados**: dá a possibilidade de estar em mais de um estado ao mesmo tempo, dividindo um estado em várias partes paralelas, chamadas de estados-AND, permitindo assim representar paralelismo e concorrência de ações.
- **Eventos primitivos**: são eventos pré-definidos nos *statecharts* para auxiliar a manipulação do fluxo de transições. Segue a lista com os eventos primitivos que o *statechart* disponibiliza segundo [HPSS87].
  - **en**(**s**) - Evento que ocorre ao entrar no estado s;
  - **ex**(**s**) - Evento que ocorre ao sair do estado s;
  - **in**(**s**) - Retorna *true* se o estado s estiver ativo;
  - **tr**(**c**) - Evento que ocorre no momento que a condição c se torna *true*;
  - **fs**(**c**) - Evento que ocorre no momento que a condição c se torna *false*;
  - **ch**(**c**) - Evento que ocorre no momento que c muda de valor;
  - **ny**(**e**) - Retorna *true* se e ainda não ocorreu;
  - **cr**(**c**) - Retorna o valor atual de c.

Assim como as máquinas de estados finitos, uma grande vantagem do *statechart* é a sua representação visual, o que facilita seu entendimento. A seguir será dada uma visão sobre como cada componente do *statechart* é representado graficamente.

2.2.2 Representação Visual

Os elementos básicos dos *statecharts* são os estados e transições. Cada estado pode ser representado por um retângulo com cantos arredondados (estados iniciais são estados apontados por uma seta sem um estado de origem) e cada transição é mostrada como uma seta direcionada rotulada com o nome de um evento. A Figura 2.3 apresenta um *statechart* formado pelos estados A e B, sendo A o estado inicial, e pela transição e, na qual ao ocorrer o evento e dispara a transição do estado A para o estado B que passa a ser o estado ativo.

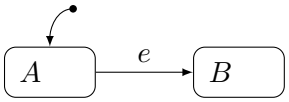
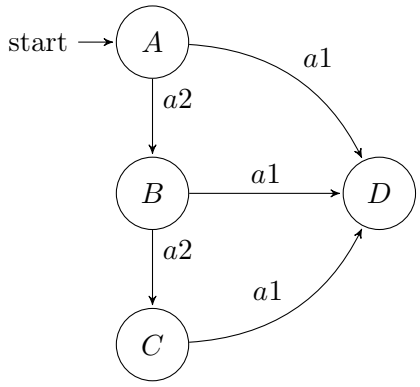


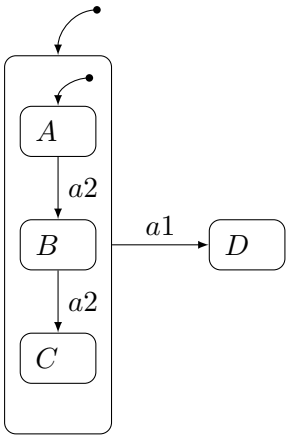
Figura 2.3: *Exemplo de estados e transições.*

As transições nos *statecharts* podem ser rotuladas no seguinte formato: $\boldsymbol{e[c]/a}$, sendo que $\boldsymbol{e}$ representa um evento que deve ocorre para disparar a transição, $\boldsymbol{c}$ é uma condição booleana necessária para que a transição aconteça, e o $\boldsymbol{a}$ é uma ação que será executada após a transição chegar no estado alvo, podendo ser uma expressão, atribuição de variável ou uma ação a ser realizada.

A hierarquia de estados ou decomposição *XOR*, pode ser representada como um estado maior com seus sub-estados internos, ou pode-se optar por omitir os estados internos, e apresentá-los em um documento diferente. A Figura 2.4b ilustra o exemplo de um *statechart* utilizando hierarquia de estados, e para demonstrar como hierarquia pode reduzir e simplificar um modelo, apresentamos ao lado, Figura 2.4a, uma máquina de estados que representa o mesmo comportamento. Neste exemplo percebe-se claramente como é possível reduzir o número de transições do modelo, além da obtenção de uma maior organização dos estados.



(a) *Exemplo de uma máquina de estados.*



(b) *Statechart equivalente a máquina de estados finitos da Figura 2.4a.*

Figura 2.4: *Exemplo de hierarquia de estados com statechart.*

A ortogonalidade ou decomposição *AND* é representada por um estado com uma linha tracejada dividindo o estado em duas ou mais partes. Cada parte passa a ser um componente *AND* do estado, trabalhando de forma independente aos outros. Na Figura 2.5 temos um exemplo de ortogonalidade.

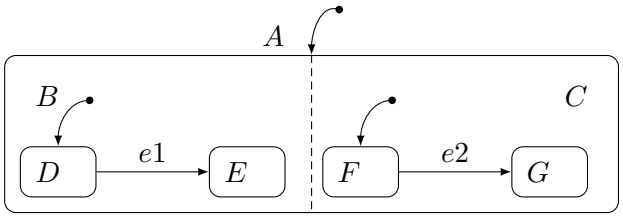


Figura 2.5: *Exemplo de ortogonalidade de estados com statechart.*

No *statechart* representado na Figura 2.5, o estado inicial será $\{D, F\}$, se ocorrer o evento $e1$, então o estado do *statechart* vai passar a ser $\{E, F\}$, ou se ocorrer o evento $e2$, teremos o estado $\{D, G\}$ como estado ativo.

Para demonstrar a capacidade do *statechart*, a Figura 2.6 apresenta um modelo *statechart* equivalente a máquina de estados da Figura 2.2. Nele é possível notar que os estados estão mais bem agrupados, e como o *statechart* consegue reduzir o número de transições (de 4 para 3) no modelo buscando simplificar o entendimento.

Como podemos ver, o *statechart* se mostra bastante rico quando se trata de descrever modelos de transições, e até por essa razão já vem sendo bastante utilizado quando se trata de especificar o comportamento desejado de um sistema. Por se tratar de um modelo já difundido no meio de desenvolvimento de software, nós optamos por utilizar o *statechart* como modelo para descrever o comportamento do software a ser observado. Contudo, como o objetivo será compará-lo com a máquina de estados finitos que representa o comportamento do software, será necessário transformar esse *statechart* da especificação em uma máquina de estados finitos para que essa comparação seja possível. A seguir será explicada quais as transformações aplicadas sobre o modelo *statechart* da especificação, a fim de que este se torne comparável com o modelo da implementação.

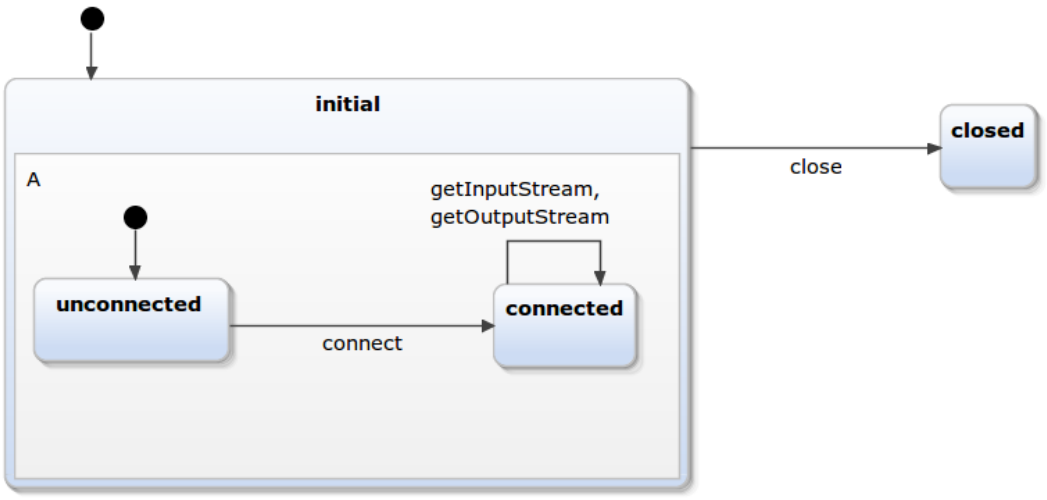


Figura 2.6: *Statechart equivalente à máquina de estados da Figura 2.2.*

2.3 Transformações do Modelo da Especificação

Sabendo que o *statechart* é o modelo utilizado como especificação neste trabalho, optou-se então por utilizar o software Yakindu [yak] como ferramenta para desenhar os modelos *statecharts* da especificação.

Como dito anteriormente, será necessário transformar o *statechart*, definido no Yakindu, em uma máquina de estados finitos para assim realizar a comparação com a implementação. Para essa tarefa será utilizado o software GTSC [SVG+08] que já implementa um algoritmo que realiza a transformação de *statechart* para máquina de estados finitos, algoritmo esse que será detalhado na Seção 2.3.2. Contudo, antes disso será necessário transformar o modelo feito no software Yakindu, em um modelo que possa ser importado pelo GTSC, e para isso será necessário realizar uma conversão de SCT, formato do software Yakindu, para PCML, formato do software GTSC. Além dessa transformação, também será necessário transformar a máquina de estados finitos obtida através do GTSC, gerada em um arquivo XML, para a instância de um objeto da classe `Automato` definida neste trabalho, que também será utilizada para representar o modelo da implementação. Portanto o processo de conversão da especificação *statechart* para um modelo na forma de uma máquina de estados finitos é realizada em três etapas distintas, sendo elas:

- **SCT2PCML**: após definir o *statechart* no software Yakindu, Apêndice A.3, é gerado um arquivo com extensão .SCT com as suas definições. Entretanto, necessitamos de um *statechart* definido no formato de PCML, formato esse que pode ser importado pelo software GTSC, software que permite converter um *statechart* em máquina de estados finitos;
- **GTSC (PCML para Máquina de Estados Finitos)**: a partir de um *statechart* definido no formato PCML, o GTSC é capaz de transformá-lo em uma máquina de estados finitos plana, seguindo as definições de Vijaykumar et al. [VCAA06];
- **XML2AUTOMATO**: a máquina de estados finitos obtida através do GTSC é gerada em um arquivo XML, e por essa razão foi será necessário converter esse XML em uma instância da classe `Automato` por nós definida, classe essa que também é utilizada para representar o modelo da implementação.

Para exemplificar cada uma dessas transformações, utilizaremos novamente a especificação *statechart* da classe `java.net.Socket` mostrada na Figura 2.6.

2.3.1 SCT para PCML

O software GTSC é capaz de realizar a conversão de um *statechart* para uma máquina de estados finitos. Para tanto, o GTSC necessita de um *statechart* definido em um formato denominado PCML, contudo, nosso *statechart* foi gerado em um arquivo no formato SCT do software Yakindu, e por isso se faz necessário converter o modelo do formato SCT para o formato PCML. Tanto o SCT como o PCML são formatos XML, mudando somente as *tags* utilizadas, portanto a transformação consiste basicamente em um mapeamento de *tags*. No SCT cada estado é representado pela *tag* `<vertices>`, a qual, por sua vez, possui em seu interior as *tags* `<outgoingTransitions>`, que definem as transições que saem deste estado, e as *tags* `<regions>`, que definem a parte interna dos estados hierárquicos ou ortogonais. A seguir apresentamos uma função de conversão relacionada a cada uma dessas *tags*.

```
convertVertices(vertices Vertices){
  if(vertices.regions.length = 0 ){      // Estado Básico
    return <state type="BASIC" name="vertices.name"/>
  }else if(vertices.regions.length = 1){ // Estado Hierárquico
    return  convertRegion(vertices.regions.get(0));
  }else if(vertices.regions.length > 1){ //Estado Ortogonal
    return <state type="AND" name="vertices.name">
        for(r : vertices.regions){
           convertRegion(r);
        }
      </state>
  }
}

convertRegion(region Region) {
        return <state type="XOR" name="region.name" Default="region.
           verticeByType('sgraph:Entry').outgoing.target">
            for(v : region.vertices){
              convertVertices(v);
            }
         </state>
}

convertEvent(transition outgoingTransitions){
    return <Stochastic Name="transition.specification" value="1.0"/>
}

convertTransition(transition outgoingTransitions){
    return <Transition Source="transition.parent.name" Destination="transition
       .target.name" Event="transition.specification" />
}
```

Definidas as funções, é montado o aquivo PCML, seguindo a seguinte estrutura:

```
regions = SCT.regions;

<?xml version="1.0" encoding="UTF-8"?>
<Pcml Title="untitledModel" Date="YYYY-MM-DD">
  <Info>
    <Author>
      <Name>Autor</Name>
    </Author>
  </Info>
  <States>
    <Root Name="top" Type="XOR">
      convertRegion(regions);
```

```
13      </Root>
14    </States>
15
16    <Events>
17      for(transition : regions.outgoingTransitions){
18         convertEvent(transition);
19      }
20    </Events>
21    <Transitions>
22      for(transition : regions.outgoingTransitions){
23         convertTransition(transition);
24      }
25    </Transitions>
26  </Pcml>
```

Tanto o modelo SCT do *statechart* da classe `java.net.Socket` que estamos utilizando como exemplo, como o modelo PCML gerado pela transformação do mesmo, podem ser vistos no Apêndice B.1.

2.3.2 *Statechart* para Máquina de Estados Finitos

O PCML gerado pode ser usado como arquivo de entrada no GTSC, pois este software permite realizar a transformação de um *statechart* numa máquina de estados finitos plana. Para efetuar esta transformação, o GTSC implementa a técnica de Vijaykumar et al. [VCAA06, VCA02]. A seguir é apresentada esta técnica, que será exemplificada por meio do *statechart* definido na Figura 2.7.

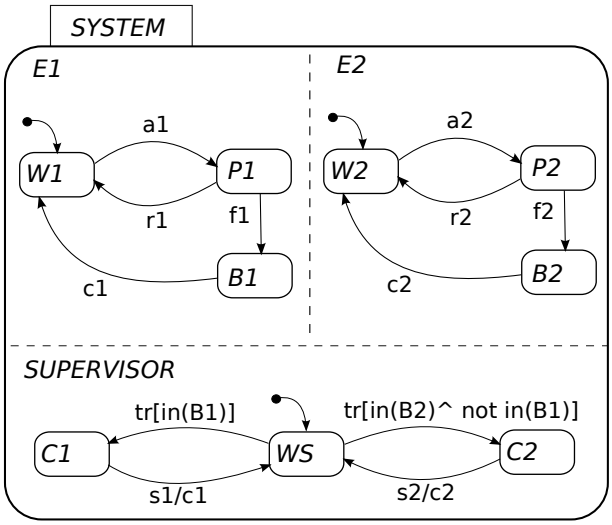


Figura 2.7: *Exemplo de statechart extraído de [VCAA06].*

Primeiramente, é necessário separar os eventos do *statechart* em duas categorias:

- **Eventos externos**, estímulos que devem ser explicitamente ativados, e
- **Eventos internos**, estímulos que são "sentidos" e disparados automaticamente, ou seja, não precisam ser explicitamente ativados. Dentre os eventos internos, estão inclusos os eventos primitivos mostrados na Seção 2.2, sendo que essa conversão somente considera os seguintes eventos primitivos [VCA02, VCAA06] [1]:
  - $en(s)$
  - $ex(s)$

[1] As ações que ocorrem como consequência de outros eventos, também serão consideradas eventos internos, como as ações $c1$ e $c2$ em nosso exemplo.

- $tr(c)$
- $fs(c)$
- $in(s)$

Em nosso exemplo, Figura 2.7, é possível categorizar os eventos da seguinte forma:

- **Eventos externos** = $\{a1, a2, r1, r2, f1, f2, s1, s2\}$;
- **Eventos internos** = $\{c1, c2, Tr(), In()\}$;

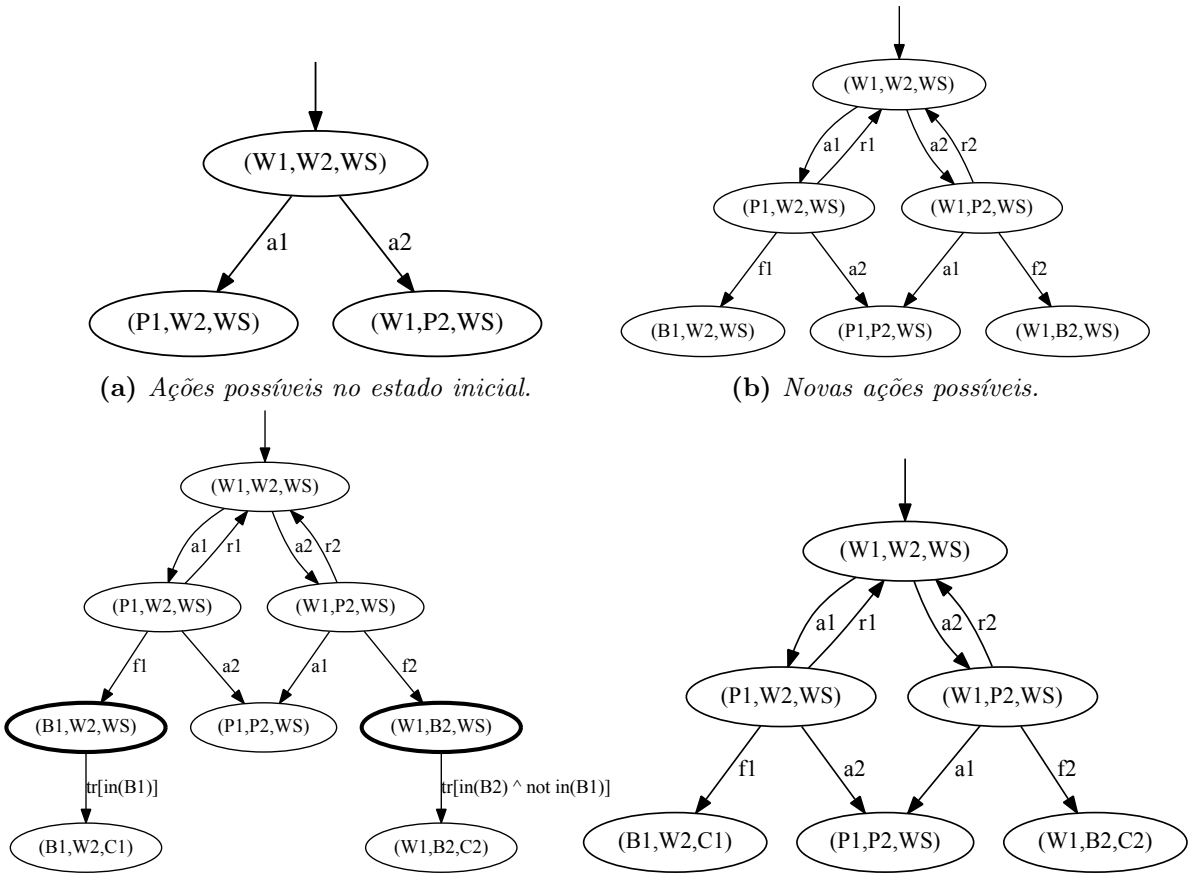
Os estados na máquina de estados finitos a ser gerada representam as configurações possíveis do *statechart*. Para iniciar a transformação é determinado o estado inicial da máquina de estados finitos, que será o estado com a configuração inicial do *statechart*. Em nosso exemplo será a configuração inicial $(W1, W2, WS)$.

- A partir da configuração inicial, é checado se existe algum evento interno que possa ser disparado nessa configuração. Como neste exemplo não existe, o passo seguinte é checar se existem eventos externos. Se existirem, eles são estimulados a fim de obter as configurações resultantes. Em nosso exemplo, há dois eventos externos, $a1$ e $a2$, conduzindo às configurações $(P1, W2, WS)$ e $(W1, P2, WS)$ respectivamente, Figura 2.8a.
- O mesmo processo é novamente aplicado nas configurações que foram obtidas, $(P1, W2, WS)$ e $(W1, P2, WS)$, o que leva a constatar que não existem eventos internos disparados nestas configurações.
- Posteriormente, temos os eventos externos $r1$ e $r2$ que irão conduzir o modelo para a configuração inicial $(W1, W2, WS)$, e os eventos $f1$ e $f2$ que conduzirão para as configurações $(B1, W2, WS)$ e $(W1, B2, WS)$ respectivamente, Figura 2.8b.
- Ao chegar nestas configurações, percebemos que os eventos internos

  $$tr(in(B1)) \text{ e } tr[in(B2) \bigwedge not\ in(B1)]$$

  serão estimulados automaticamente, levando às configurações $(B1, W2, C1)$ e $(W1, B2, C2)$, Figura 2.8c. Devido ao fato desses eventos internos serem disparados de maneira automática, no momento em que chegamos às configurações $(B1, W2, WS)$ e $(W1, B2, WS)$, não haverá outras opções de eventos a serem executadas nelas, e por isso tais configurações são omitidas da máquina de estados finitos, como podemos ver na Figura 2.8d.
- O processo seguirá com as configurações resultantes até que todo o *statechart* tenha sido percorrido, Figura 2.9.

Nesta seção foi utilizado o próprio *statechart* do trabalho de Vijaykumar et al. [VCA02] [VCAA06] como exemplo para podermos explicar melhor a transformação, visto que é um exemplo mais completo por possuir eventos internos e ortogonalidade de estados. Voltando ao nosso exemplo da especificação da classe `java.net.Socket`, a máquina de estados finitos obtida após a conversão pode ser vista na Figura 2.10, e o XML no qual o GTSC gerou essa máquina de estados finitos pode ser vista no Apêndice B.2.



(a) *Ações possíveis no estado inicial.*

(b) *Novas ações possíveis.*

(c) *Eventos Internos são disparados.*

(d) *Eliminação de estados.*

Figura 2.8: *Exemplo de derivação de um statechart.*

2.3.3 XML para Automato

Por fim, a máquina de estados finitos gerada pelo GTSC está no formato de um XML, e por esse motivo é realizada outra conversão para que o XML se torne uma instância do objeto da classe `Automato`, Listagem 2.1, definida neste trabalho, sendo esse o mesmo utilizado nos modelos minerados do programa.

```
1  class Automato {
2      ArrayList<State> states;
3      ArrayList<Transition> transitions;
4  }
5  class State {
6      String name;
7      isInitial boolean;
8  }
9  class Transition {
10     State incoming;
11     State outgoing;
12     String event;
13 }
```

Listagem 2.1: *Definição das classes Automato, State e Transition.*

A conversão do XML para a classe `Automato` é bem simples, visto que a estrutura do mapeamento XML é bem semelhante à estrutura da nossa classe. Na Listagem 2.2 mostramos como transformar as *tags* XML nos atributos da classe `Automato`.

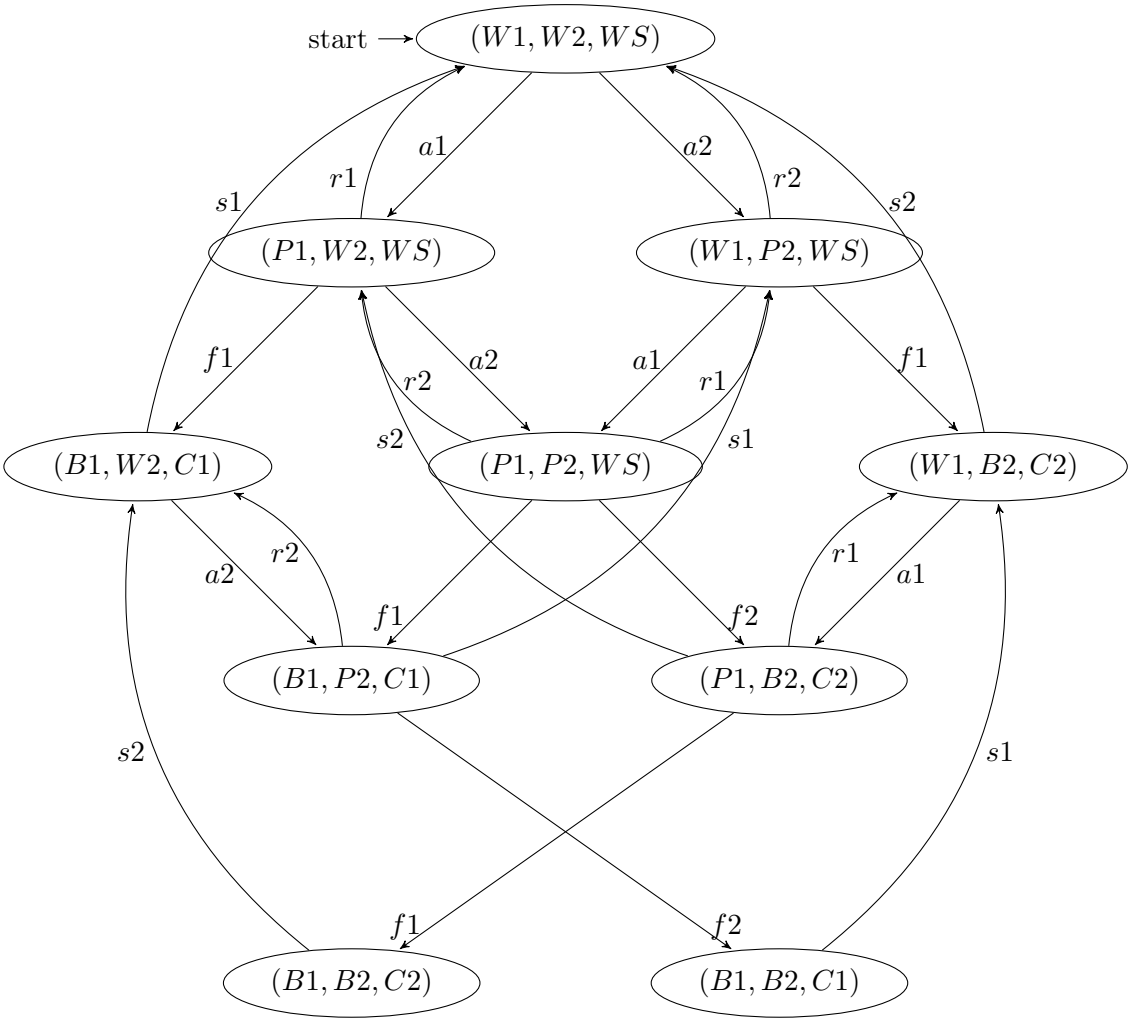


Figura 2.9: *Máquina de estados finitos resultante da transformação do statechart da Figura 2.7, extraído de [VCAA06].*

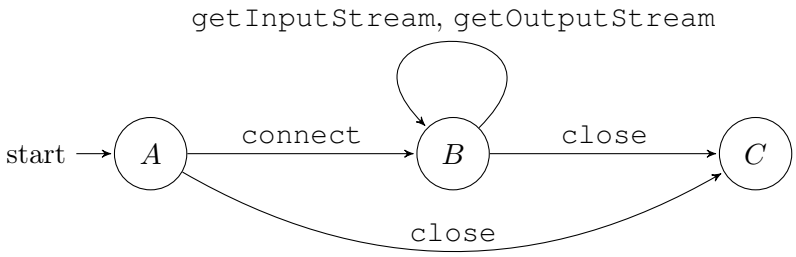


Figura 2.10: *Máquina de estados referente a especificação da classe Socket.*

```
for(state : XML.states){
  State s = new State();
  s.name = state.name;
  s.initial = state.type.equal("inicial");
  Automato.add(s);
}

for(transition : XML.transitions){
  Transition t = new Transition();
  t.outgoing = transition.destination;
  t.incoming = transition.source;
```

```
12    t.event = transition.input;
13    Automato.transitions.add(s);
14  }
```

Listagem 2.2: *Conversão do XML para objeto* `Automato`.

2.4 Conclusão

Os *statecharts* são modelos bastante utilizados para especificar comportamento de softwares por seu poder de representação e sua simplicidade. Neste trabalho supõe-se que já exista um *statechart* na documentação do software que descreva todo o possível comportamento que pode ser executado por um objeto que se deseja observar. Contudo, sabendo que o modelo obtido a partir da implementação, por simplicidade, se encontra no formato de uma máquina de estados finitos, foi necessário então transformar esse *statechart* em uma máquina de estados finitos, para que assim eles pudessem ser comparados. No próximo capítulo serão apresentados os conceitos e definições utilizados para obter essa máquina de estados finitos que representa o comportamento de um objeto do software, a partir do sistema já implementado.

Capítulo 3

Modelo da Implementação

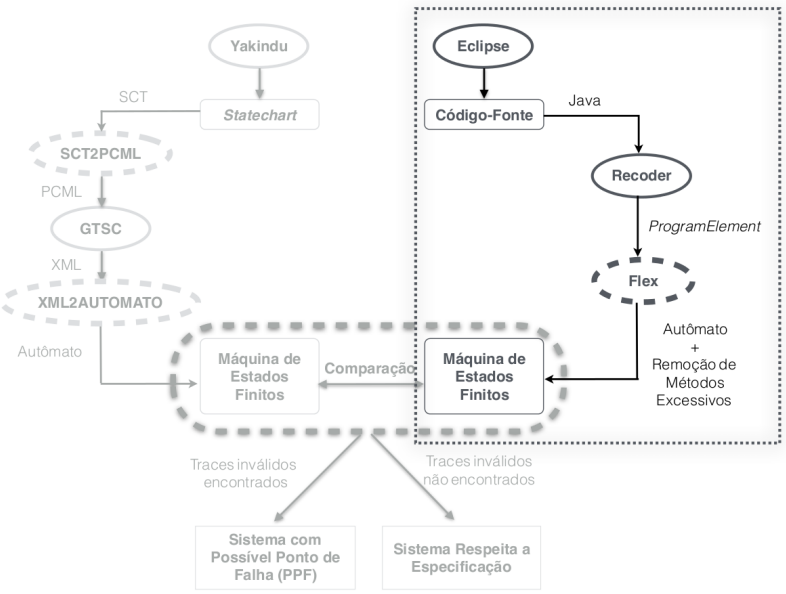


Figura 3.1: *Fluxo completo da abordagem (Implementação).*

Após ter a especificação já no formato de uma máquina de estados finitos, o próximo passo é obter o modelo que represente a implementação do software. Neste capítulo apresentamos uma visão geral sobre mineração de especificação e como essa técnica será utilizada para minerar uma máquina de estados finitos a partir de um código-fonte Java. Além disso serão mostradas as modificações necessárias nesta máquina de estados finitos minerada do sistema, para facilitar a comparação com a especificação posteriormente.

3.1 Mineração de Especificação

O processo de desenvolvimento e manutenção de software tem sofrido cada dia mais por falta de documentação nos sistemas. Essa falta de documentos que descrevam as funcionalidades do software faz com que as chances de haver *bugs* no sistema se tornem maiores, visto que não se dispõe de um documento que mostre exatamente o que o sistema deve fazer. Outro fator agravado pela falta de documentação é o custo de manutenção, pois este se torna maior devido ao fato dos desenvolvedores consumirem mais tempo para entender e modificar o sistema. Existem estudos que mostram que o custo de manutenção do software pode chegar a ser 90% do custo total do sistema [LKHL11]. A importância dos documentos que especifiquem o sistema e a realidade de que nem sempre esses documentos são feitos ou estão atualizados, motivaram o crescimento do estudo de mineração de especificação, dado que essa técnica permite obter modelos de especificação a partir do sistema já

implementado.

A mineração de especificação, *Specification Mining* (SM), pode ser definida como a tarefa de obter uma especificação a partir do software já implementado. Dentre as informações que podem ser obtidas a partir do software, as que mais se destacam são [LKHL11]:

- Máquinas de estados: permitem saber os possíveis fluxos de chamadas de métodos que o sistema está implementando;
- Padrões: identificam padrões de chamadas de métodos em diferentes partes do sistema;
- Invariantes: possibilitam saber restrições a determinadas variáveis locais ou globais no sistema;
- Diagrama de sequência: permite saber o ciclo de vida de alguma informação do software, e ver como partes diferentes do software se comunicam e tratam essa informação.

Essas informações obtidas através de mineração de especificação podem ser utilizadas com diversos propósitos pelo desenvolvedor, como:

- Verificar a corretude do software: a especificação minerada pode ser comparada com a especificação inicial do software, no intuito de verificar a fidelidade do software em relação a especificação inicial do sistema;
- Atualizar a especificação: em alguns momentos da vida da aplicação a especificação fica defasada em relação ao sistema, então uma especificação extraída, a qual representa a situação atual do software, pode ser usada como nova especificação;
- Facilitar o entendimento do funcionamento do software: dado que um modelo extraído permite uma representação mais simples e de mais fácil entendimento do que tentar buscar as informações diretamente no código-fonte.

Como visto, essas informações possuem diversas utilidades, e podem auxiliar o desenvolvedor, reduzindo tempo e custo do projeto. Essas informações podem ser obtidas de duas maneiras, a partir da observação do software em funcionamento, **análise dinâmica**, ou a partir da observação do código-fonte, **análise estática**.

Através da análise dinâmica, isto é, da mineração por observação do sistema em funcionamento, é possível observar o sistema em execução em cenários pré-definidos e assim obter informações de como ele se comporta naquelas situações. Isso possibilita a análise de maiores detalhes do comportamento real do sistema, entretanto, a técnica fica limitada por não cobrir todas as possibilidades de execução que podem ocorrer no software, cobrindo apenas o que é executado durante a observação.

Por outro lado, a mineração estática analisa o código-fonte sem a necessidade do software ser executado. Com ela é possível observar o cenário como um todo, independente do contexto, através das observações das estruturas de controle, chamadas de métodos e declaração de variáveis. Todavia, essa abordagem não permite a obtenção de informações como: valores de variáveis e resultados de operações lógicas no código.

Sabendo disso podemos concluir que as duas abordagens possuem vantagens e desvantagens, podendo ser usadas de acordo com a necessidade, ou até mesmo em conjunto, de forma que as informações obtidas por uma técnica possam complementar a outra. Neste trabalho optou-se pelo uso da análise estática, pois ela oferece uma maior cobertura na obtenção dos possíveis fluxos de chamada de métodos e é de fácil manuseio.

Neste trabalho utilizamos a mineração de especificação não para obter uma especificação do software, mas para obter um modelo que represente o comportamento já implementado do software.

Para realizar essa mineração, desenvolvemos o algoritmo Flex, o qual observa códigos-fonte escritos em Java, para construir uma máquina de estados finitos que represente todos os possíveis fluxos que podem ocorrer com um objeto em uma determinada parte do software. Os critérios utilizados pelo algoritmo Flex para a montagem dessa máquina de estados finitos serão apresentados na seção a seguir.

3.2 Algoritmo Flex para Mineração de Especificação

Para obter o modelo da implementação do software, nós utilizamos mineração de especificação sobre um sistema escrito na linguagem Java, buscando construir uma máquina de estados finitos do comportamento do software. Para realizar essa mineração, utilizamos o *framework* Recoder [rec], Apêndice A.2, que analisa o código-fonte do sistema, e provê as estruturas de controle e os comandos do sistema em forma de instâncias do objeto `recoder.java.ProgramElement`. Através desses objetos e de um conjunto de definições para cada tipo de estrutura de controle, nós implementamos o algoritmo Flex, o qual cria a máquina de estados finitos que representa os possíveis comportamentos de um objeto específico no software. Optou-se por observar somente um objeto durante a mineração por questão de simplicidade.

3.2.1 Extração de uma Máquina de Estados Finitos a partir do Código-Fonte

Ammann e Offutt [AO08] apresentam critérios de cobertura de software que possibilitam representar o comportamento implementado de um software. Dentre eles, temos uma técnica denominada **Cobertura Gráfica Estrutural para Código-Fonte**, na qual são utilizados grafos direcionados para representar as estruturas de controle contidas no código-fonte. Esta técnica pode apresentar variações dependendo da linguagem de programação na qual for aplicada. Porém seus conceitos básicos são válidos para as linguagens de programação mais populares, visto que as principais estruturas de controle (`For, While, If-Else` e `Switch`) são semelhantes na maioria das linguagens atuais.

Neste mesmo trabalho, Ammann e Offutt [AO08] argumentam que antes de se iniciar o processo de cobertura do código é necessário definir:

- qual modelo de grafo será usado; e
- qual estrutura de código será observada.

Logo, para aplicação da técnica dada por Ammann e Offutt [AO08], escolhemos adaptar o conceito de **Grafo de Fluxo de Controle** (GFC), onde construiremos uma máquina de estados finitos em que cada aresta representa um possível caminho no fluxo do sistema e cada estado representa um ponto do sistema após a execução de um comando. Com essa abordagem, podemos representar todos os possíveis fluxos de comandos que podem ocorrer em um cenário do software.

Por fim, optamos por gerar modelos baseados em códigos-fonte escritos na linguagem de programação Java, por

- ser uma das linguagens mais utilizadas para o desenvolvimento de software na atualidade; e
- possuir uma sintaxe simples e semelhante a de outras linguagens bem conhecidas como *C++* e *C#*, o que faz com que as técnicas desenvolvidas aqui para a linguagem Java, possam ser adaptadas para essas outras linguagens.

A partir dessas definições, adaptaremos a técnica de cobertura gráfica estrutural para código-fonte proposta por Ammann e Offutt [AO08] na criação de uma máquina de estados finitos com os fluxos de processos de um objeto em um software codificado na linguagem Java. Diferentemente do trabalho de Ammann e Offutt, que considera todos os comandos no código, neste trabalho optamos inicialmente por considerar somente as estruturas de controle principais, `If-Else`, `Switch`, `For` e `While`, e os comandos em que há manipulação direta de um objeto a ser previamente especificado pelo desenvolvedor. Para efeito do comportamento do software, entendemos que um objeto é manipulado quando:

- o objeto realiza uma chamada de método; ou
- o objeto é passado como parâmetro para algum método.

A técnica de cobertura gráfica estrutural para código-fonte por nós utilizada, foi uma adaptação nossa dos conceitos de Ammann e Offut, e permite cobrir as principais estruturas de controle (`If-Else`, `While`, `For`, `Switch`) contidas em um código-fonte escrito em linguagem Java, e dessa forma definir todos os possíveis fluxos de chamadas de métodos que poderão ocorrer com um objeto quando o software for executado. Para que estes resultados sejam obtidos, é necessário realizar um processo de montagem do grafo. Este processo possui particularidades para cada estrutura de controle que se deseja modelar e será explicado em detalhes a seguir. É importante ressaltar que o processo utiliza um algoritmo recursivo o qual não possui limitações quanto a estruturas complexas, como por exemplo, um `If-Else` aninhado dentro de um `While`.

O mapeamento do código-fonte para o grafo é feito selecionando comandos simples, como a chamada de um método. Para cada um dos comandos selecionados, é inserida uma aresta que realiza a transição entre um nó que já existe no grafo e um novo nó a ser inserido. Esta aresta é rotulada com o nome do comando. Este novo nó, por sua vez, representa um novo estado do programa que é atingido após a execução do comando selecionado. Por fim, a sequência de transições representa os possíveis fluxos de controle do objeto a ser observado, como pode ser visto no exemplo da Figura 3.2, onde temos a máquina de estados finitos obtida a partir da observação do objeto `s`.

```
1 public void init(){
2   s.getOutputStream();
3 }
```

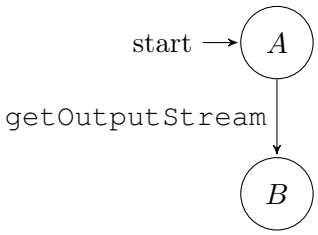


Figura 3.2: *Exemplo de máquina de estados finitos para chamada de métodos.*

Como essas chamadas de métodos, Figura 3.2, são geralmente agrupados em estruturas de controle, onde cada uma dessas estruturas pode alterar o fluxo de execução do software de uma maneira diferente, é necessário definir padrões para representar cada umas dessas estruturas dentro de um modelo de transição, ao invés de só considerar os comandos individuais. Sendo assim, para cada uma dessas estruturas de controle é necessário utilizar um algoritmo específico que permita representar essa estrutura no formato de uma máquina de estados finitos. Tais algoritmos são apresentados na Seção 3.2.2.

3.2.2 Modelagem das Estruturas de Controle

Para a construção do grafo que representa as estruturas de controle, é necessário que estas sejam entendidas como blocos de comando. Cada bloco representa um conjunto de comandos que

ao serem executados comutam o software de um estado para outro. Tomando como base os conceitos de Ammann e Offutt [AO08], decidimos então analisar as seguintes estruturas de controle:

- `if-else`;
- `if`;
- `while`;
- `for`[1], e
- `switch`.

A construção do grafo que representa o bloco `if-else` tem início com a inserção de um estado denominado estado de decisão, a partir do qual o grafo pode seguir dois caminhos, que são:

- o bloco `if` é executado, ocorre quando a decisão booleana do comando `if` é verdadeira; e
- o bloco `else` é executado, ocorre quando a decisão booleana do comando `if` é falsa.

Por fim, é necessário unir os dois fluxos novamente, para isso, é adicionado um novo estado, estado de junção, que será conectado aos fluxos por transições vazias. Essas transições vazias, que também serão utilizadas nas outras estruturas de controle, significam que não há a necessidade de nenhuma entrada para avançar de um estado para o outro. Um exemplo dessa transformação pode ser visto na Figura 3.3, quando o objeto `s` é observado.

```
1 public void close(SocketAddress
    endpoint){
2     if(canClose()){
3         s.close();
4     }else{
5         s.connect(endpoint);
6         s.getOutputStream();
7     }
8 }
```

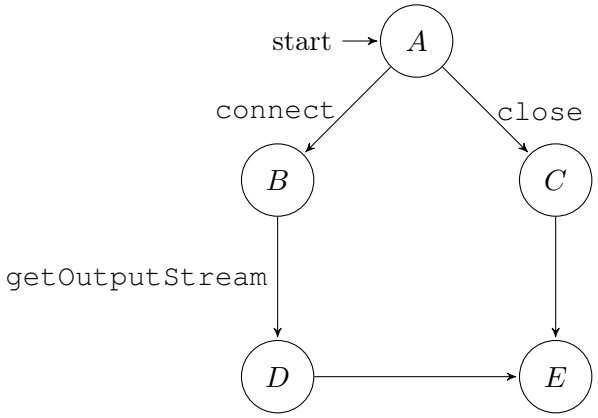


Figura 3.3: *Exemplo de máquina de estados finitos para um bloco `if-else`.*

Se considerarmos apenas o bloco `if`, sem o bloco `else`, obtemos a construção de um grafo que se assemelha ao grafo do bloco `if-else` apresentado anteriormente, no qual é adicionado um estado de decisão no início do grafo com um fluxo que representa os processos dentro do `if`. Porém, neste caso, há um caminho que simplesmente aponta para o estado final, representando a situação em que a decisão do comando `if` não é satisfeita, e portanto não há comandos a serem executados. Um exemplo dessa transformação pode ser visto na Figura 3.4, quando se busca o comportamento do objeto `s`.

Para representar um bloco `while`, é necessário criar um estado de decisão semelhante ao do `if`. Este estado de decisão possui duas transições possíveis, uma para a situação na qual a decisão do `loop` é satisfeita, decisão valorada `true`, e outra transição para quando a decisão do `loop` não é satisfeita, decisão valorada `false`. Se a transição que satisfaz a decisão ocorrer então o fluxo de controle da aplicação executa o código do bloco `while`, caso contrário, o fluxo de controle é

[1] O bloco `for` seguirá os mesmos critérios do bloco `while`.

```
1 public void size(){
2   if(i < 0){
3     s.close();
4   }
5 }
```

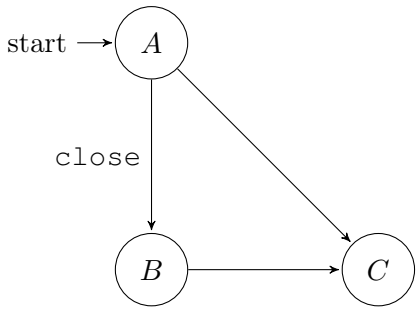


Figura 3.4: *Exemplo de máquina de estados finitos para um bloco `if` sem o bloco `else`.*

desviado para um estado final. Por fim, é adicionada uma transição que aponta para o estado de decisão, representando a volta do `loop`. Um exemplo desse tipo de transformação pode ser visto na Figura 3.5, quando se observa o objeto `s`.

```
1 public void read(){
2   int i = 0;
3   while (i < 10) {
4     s.getInputStream();
5     i++;
6   }
7 }
```

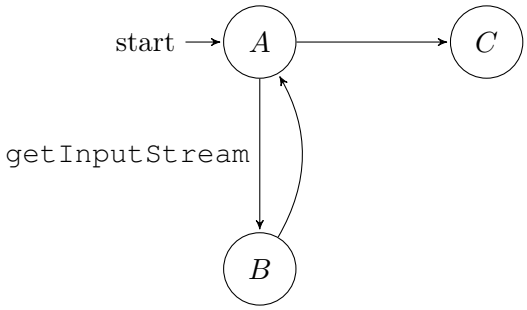


Figura 3.5: *Exemplo de máquina de estados finitos para um `loop`.*

Por último, para a construção do grafo de um bloco `switch`, é necessária a inserção de um estado de decisão, a partir do qual será montado um possível fluxo de controle para cada um dos `cases` do comando `switch`, incluindo o `case default`. Por fim, é adicionado um estado final de junção no qual todos os fluxos retornam para um mesmo estado, como pode ser visto no exemplo da Figura 3.6, onde o objeto `s` é o objeto observado.

```
1  public void menu(){
2    String opc = new String();
3    read(opc);
4      switch (opc) {
5      case 'O':
6          s.getOutputStream();
7          break;
8      case 'C':
9          s.close();
10         break;
11     case 'P':
12         s.getPort();
13         break;
14     }
15 }
```

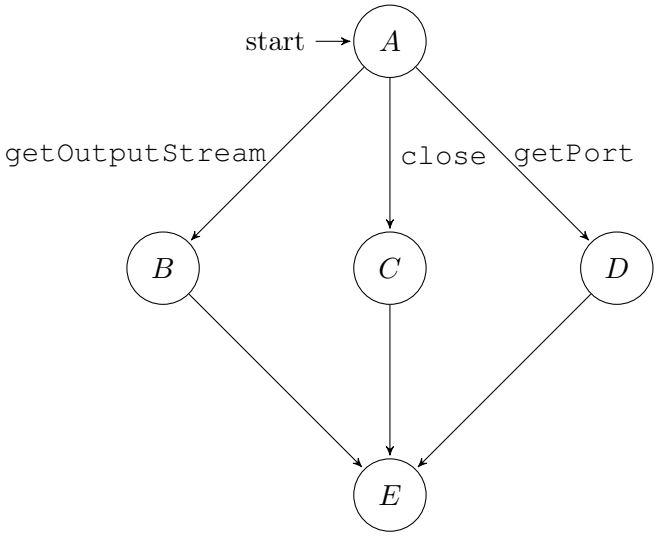


Figura 3.6: *Exemplo de máquina de estados finitos para um `switch`.*

Com essas definições, para cada estrutura de controle é possível construir uma máquina de estados finitos para representar o possível comportamento do objeto. Como o objetivo é somente

o comportamento de um objeto específico, outras ações que existam no código, que não tenham relação com o objeto que está sendo observado serão desconsideradas.

3.2.3 Algoritmo Flex para Geração do Grafo

As definições dos métodos de transformação das estruturas de controle da Seção 3.2.2, inspiradas nas definições dadas por Ammann e Offutt [AO08], possibilitou que nós definíssemos e implementássemos o **Algoritmo Flex** (*Flow Extractor*), Algoritmo 1. Este algoritmo tem como entrada um código-fonte Java e o objeto que deverá ser observado. A partir disso, o algoritmo observa onde o objeto pode ser utilizado e como esse objeto é manipulado dentro das estruturas de controle. Através dessa observação do objeto, o algoritmo constrói e retorna uma máquina de estados finitos que representa o possível comportamento do objeto observado dentro do sistema.

O algoritmo Flex realiza uma análise estática do sistema, ou seja, observa o código-fonte, e não o sistema em execução. Por se tratar de uma análise das estruturas do código, definimos quais dessas estruturas e comandos seriam considerados no momento da "varredura" do código-fonte. Na Tabela 3.1 vemos os comandos e estruturas que iremos observar ao analisar o código, as outras estruturas ou comandos, como *try-catch*, *do-while*, ou atribuições diretas aos objetos não serão observadas por enquanto.

Tipo	Descrição	Exemplo
While	Estrutura de repetição While.	`while(n>1){...}`
For	Estrutura de repetição For.	`for(i=0;i<10;i++){...}`
Switch	Estrutura de decisão Switch.	`switch(a){...}`
If	Estrutura de decisão If.	`if(n>1){...}`
If-Else	Estrutura de decisão If-Else.	`if(n>1){...}else{...}`
MethodCall	Objeto realizando uma chamada de método.	`object.toString();`

Tabela 3.1: *Tipos de comandos observados pelo algoritmo Flex.*

Como pode ser visto no Algoritmo 1, o algoritmo Flex recebe uma lista de comandos, `comandos`, que é o código-fonte a ser analisado, e o objeto a ser observado, `objeto`. Esses comandos podem ser estruturas de controle ou uma manipulação no objeto (objeto chamando um método, ou sendo passado como parâmetro).

O algoritmo Flex irá percorrer toda a lista de comandos, linha 4, para construir a máquina de estados finitos referente ao uso do objeto observado. Percorrendo a lista de comandos, o Flex verifica o tipo do comando, e se utilizando das definições explicadas anteriormente na Seção 3.2.2, ele constrói a máquina de estados finitos. A seguir vemos qual parte do algoritmo é referente a cada estrutura.

- Linha 5 - `while-for`: se o comando for um bloco `while`, o primeiro passo é construir a máquina de estados finitos do conteúdo dessa estrutura, linha 6. Após isso, serão adicionados a essa máquina de estados finitos o estado de junção e as transições vazias, linha 7, e por fim essa máquina de estados finitos é adicionada no final da máquina de estados finitos principal, `mef.adicionar`.
- Linha 9 - `switch`: se o comando for um bloco de decisão `switch`, o primeiro passo é construir a máquina de estados finitos do conteúdo dessa estrutura, linha 10, e logo após, serão adicionados à máquina de estados finitos o estado de junção e as transições vazias, linha 11, e por fim essa máquina de estados finitos é adicionada a máquina de estados finitos principal, `mef.adicionar`.

Algoritmo 1: Algoritmo Flex

```
   Entrada: comandos, objeto
   Saída: Maquina de Estados Finitos
1  gerarGFC (comandos, objeto)
2  início
3      mef ← [ ] ;
4      para cada comando em comandos faça
5          se comando é While ou comando é For então
6              mefWhile ← gerarGFC(comando.interno(), objeto);
7              mef.adicionar(gerarGFCWhile(mefWhile));
8          fim
9          se comando é Switch então
10             mefSwitch ← gerarGFC(comando.interno(), objeto);
11             mef.adicionar(gerarGFCSwitch(mefSwitch));
12         fim
13         se comando é If então
14             mefIf ← gerarGFC(comando.interno(), objeto);
15             se comando tem Else então
16                 mefElse ← gerarGFC(comando.else.interno(), objeto);
17                 mef.adicionar(gerarGFCIfElse(mefIf, mefElse));
18             fim
19             senão
20                 mef.adicionar(gerarGFCIf(mefIf));
21             fim
22         fim
23         se ((comando é ChamadaDeMetodo) e (comando modifica objeto)) então
24             mef.adicionaTransicao(comando);
25         fim
26     fim
27     retorna mef;
28 fim
```

- Linha 13 - `if-else`: se o comando for um bloco de decisão `if` ou um `if-else`, o primeiro passo é construir a máquina de estados finitos do conteúdo que está dentro do `if`, linha 14. Após isso é verificado se este `if` possui um `else`, se sim, é gerada a máquina de estados finitos do conteúdo do `else`, linha 16, e logo em seguida as duas máquinas de estados finitos são mescladas, linha 17. Se não houver o bloco `else`, é só gerar a máquina de estados finitos referente ao `if` sem o `else`, linha 20, enfim essa máquina de estados finitos é adicionada ao final da máquina de estados finitos principal, `mef.adicionar`.

- Linha 23 - `chamadaDeMetodo`: se o comando for uma chamada de método, a primeira coisa a fazer é verificar se essa chamada de método está relacionada ao objeto observado, `comando modifica objeto`, se sim, então essa chamada de método é transformada em uma transição na máquina de estados finitos, `mef.adicionaTransicao(comando)`.

3.2.4 Exemplo

Para melhor entendimento dos parâmetros de entrada do Algoritmo Flex, o exemplo da Listagem 3.1 ilustra um código a ser observado pelo Algoritmo Flex e, ao lado, na Tabela 3.2, temos a lista de comandos que será considerada pelo algoritmo, representando os comandos contidos no código-fonte de entrada. Os comandos internos das estruturas de controle não são passados ex-

plicitamente, porém são obtidos através do método `interno()` aplicado sobre o comando que representa a estrutura, como pode ser visto nas linhas 6, 10, 14 e 16 do Algoritmo 1.

A saída do algoritmo Flex será a máquina de estados finitos que corresponde às possíveis manipulações do objeto a ser observado, cobrindo todas as possibilidades de fluxo de ações que podem ocorrer com este objeto. Para exemplificar, na Figura 3.7 temos a máquina de estados finitos gerada pelo Algoritmo Flex a partir da observação do objeto `socket` da Listagem 3.1. Partindo da premissa de que existe uma especificação *statechart* para esse objeto, é possível então realizar uma verificação que garanta que todas as mudanças de estado do objeto são previstas na especificação.

```
1  public void method(){
2    socket.connect(addressPoint);
3    while (i < 10) {
4      i++;
5      if(i % 2 == 0){
6        socket.getInputStream();
7      }
8      socket.getPort();
9    }
10   socket.close();
11 }
```

Listagem 3.1: *Exemplo de método.*

Linha	Código	Tipo
2	socket.connect(...)	MethodCall
3	while (i < 10) {...}	While
10	socket.close()	MethodCall

Tabela 3.2: *Comandos passados ao algoritmo Flex.*

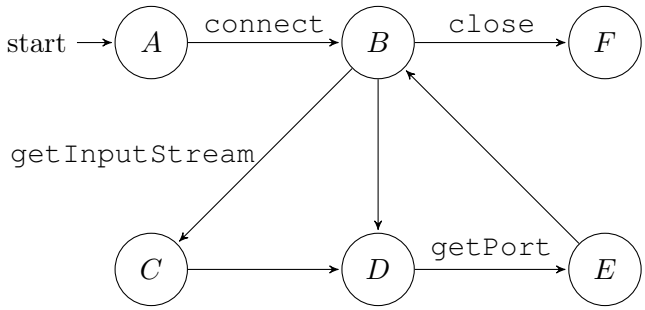


Figura 3.7: *Máquina de estados finitos gerada pelo algoritmo Flex para a Listagem 3.1.*

3.3 Transformações do Modelo da Implementação

Após o algoritmo Flex minerar a máquina de estados finitos com o possível comportamento do objeto, ainda será necessário fazer algumas alterações neste modelo para que a comparação com a especificação seja feita.

Nós entendemos que há métodos que devam existir somente na implementação do software, sendo assim não necessitam estar descritos na especificação. Por esta razão, em nossa abordagem serão considerados somente os métodos definidos no modelo da especificação. Por essa razão, omitiremos os métodos que só existam no modelo da implementação, mas que não façam parte da especificação. Σ_{sub}, Definição 1, denota o conjunto de eventos da implementação que deverão ser substituídos por uma ação não observável (τ).

Definição 1 *Sejam* $E = \{Q_E, \Sigma_E, \delta_E, qE_0\}$ *e* $I = \{Q_I, \Sigma_I, \delta_I, qI_0\}$ *duas máquinas de estados finitos que representem o modelo da especificação e da implementação respectivamente. O alfabeto de métodos a serem ignorados no modelo da implementação é definido por:*

- $\Sigma_{sub}(I, E) = \{m \mid m \in (\Sigma_I - \Sigma_E)\}$

Sabendo o alfabeto de métodos a serem ignorados, dado pela Definição 1, utilizamos então o Algoritmo 2 para transformar as transições da implementação que são rotuladas com os eventos contidos no alfabeto Σ_{sub} em transições τ.

Algoritmo 2: Transformar métodos τ

Entrada: $I = \{Q_I, \Sigma_I, \delta_I, q0_I\}$ e $E = \{Q_E, \Sigma_E, \delta_E, q0_E\}$

```
1 início
2     para ∀(q --w--> q') ∈ δ_I faça
3         se w ∈ Σ_sub(I, E) então
4             substitua w por τ
5         fim
6     fim
7 fim
```

Para exemplificar, imagine que o algoritmo Flex minerou a máquina de estados finitos representada na Figura 3.8. Nela podemos ver o método `getPort`, que não está descrito em nossa especificação, como pode ser visto na Figura 3.9, portanto esse método será transformado em uma transição τ, como visto da Figura 3.10.

Figura 3.8: *Exemplo de uma máquina de estados finitos minerada pelo algoritmo Flex.*

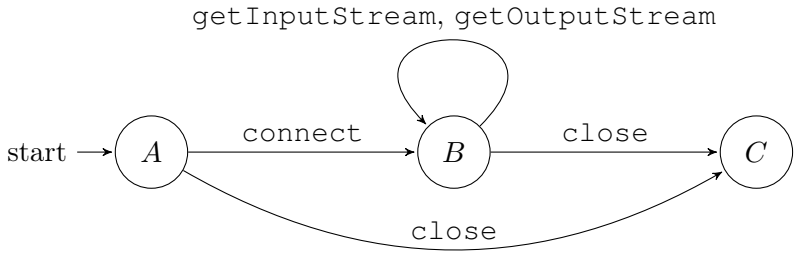


Figura 3.9: *Máquina de Estados referente a especificação da classe* `Socket`.

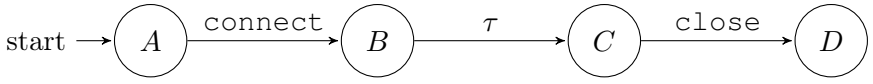


Figura 3.10: *Máquina de estados finitos da Figura 3.8 após a omissão dos métodos desconhecidos da especificação.*

Após essas modificações, o modelo da implementação segue a seguinte definição:

Definição 2 *Sejam* $E = \{Q_E, \Sigma_E, \delta_E, qE_0\}$ *e* $I = \{Q_I, \Sigma_I, \delta_I, qI_0\}$ *duas máquinas de estados finitos que representem o modelo da especificação e da implementação respectivamente. A máquina de estados finitos* $I_\tau = \{Q_I, \Sigma_\tau, \delta_\tau, qI_0\}$ *representará a máquina de estados finitos da implementação após a omissão dos eventos.*

- Q_I*: Conjunto não vazio de estados;*
- $\Sigma_\tau = \{(\Sigma_I - \Sigma_{sub}(I, E)) \cup \tau\}$
- δ_τ*:* $Q_I \times \Sigma_\tau \rightarrow Q_I$ *é a função de transição de estados;*
- $q0_I \in Q_I$ *é o estado inicial;*

3.4 Conclusão

A máquina de estados finitos gerada pelo algoritmo Flex representa todo o possível comportamento que pode ser executado pelo objeto quando o software for executado. Após a alteração feita para que esta máquina de estados finitos fique com o mesmo alfabeto de métodos da especificação, a máquina de estados finitos da implementação pode ser então comparada com a especificação, definida no capítulo anterior. No capítulo seguinte será explicada o tipo de relação utilizada para comparar o modelo da implementação e o da especificação, e como essa relação permite identificar os possíveis pontos de falhas no software ou garantir que o software realmente implementa a sua especificação.

Capítulo 4

Comparação da Implementação com a Especificação

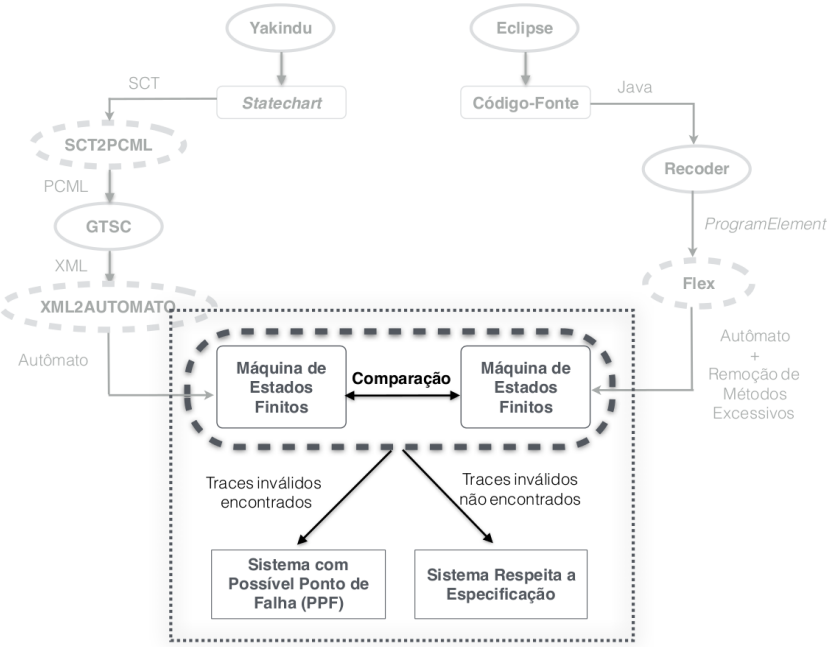


Figura 4.1: *Fluxo completo da abordagem (Comparação dos Modelos).*

Com os modelos da especificação e da implementação já no formato de máquinas de estados finitos, o objetivo agora é estabelecer uma relação de similaridade entre eles e verificar se a implementação está realmente obedecendo a sua especificação. Uma maneira de checar se a implementação obedece sua especificação, é mostrar que ambos os modelos representam os mesmos *traces*[1], e portanto a implementação reflete o comportamento da especificação. Contudo, é preciso considerar que nem sempre é necessário que ambos descrevam exatamente os mesmos *traces*, como nos casos onde a especificação descreve o comportamento que **pode** ser executado, ao invés do comportamento que **deve** ser executado [GLLS07, SG03]. Nestes casos, é necessário que a implementação obedeça as regras definidas na especificação, porém sem a necessidade de implementar tudo que a especificação descreve. Como neste trabalho parte-se da ideia de que o *statechart* descreve o comportamento que **pode** ser realizado, a relação de similaridade a ser estabelecida será de que os *traces* do modelo da implementação devem estar todos descritos na especificação, porém, não é necessário que todos os

[1]Neste trabalho consideramos o *trace* como sendo uma sequência de ações (chamada de métodos) que podem ocorrer com o objeto

traces da especificação estejam na implementação.

Dentre as várias relações de similaridades que podem ser feitas entre máquinas de estados finitos, temos a *Trace Inclusion* [BK+08], que é uma relação de ordem parcial que se mostrou capaz de verificar a relação buscada neste trabalho. A seguir, serão explicados os conceitos de *trace equivalence*, que é uma relação de similaridade onde os *traces* da implementação devem ser iguais aos *traces* da especificação, e também será explicado a *trace inclusion*, onde os *traces* da implementação devem estar todos descritos na especificação, porém a especificação pode ter outros *traces* que não estejam na implementação.

4.1 *Trace Equivalence*

Trace Equivalence é uma relação de similaridade que pode existir entre dois estados, ou entre duas máquinas de estados finitos. Podemos dizer que dois estados, q e q', são *trace* equivalentes se ambos descrevem os mesmos *traces*, $Traces(q) = Traces(q')$. Para saber se duas máquinas de estados finitos, TS e TS', são *trace* equivalentes, basta saber se seus estados iniciais, ts_0 e ts'_0, são *trace* equivalentes, $Traces(ts_0) = Traces(ts'_0)$ [TPBL95].

Para entender como funciona o *trace equivalence*, é necessário entender o conceito de Propriedades de Tempo Linear. Propriedades de tempo linear (propriedades TL), do inglês *Linear-Time Properties*, especificam os *traces* que uma estrutura deve exibir, em outras palavras, podemos dizer que uma propriedade de tempo linear especifica o comportamento admissível (ou desejável) de um sistema [LL13]. Sabendo disso, se dois sistemas de transições possuem os mesmos *traces*, então eles satisfazem as mesmas propriedades TL. Quando duas máquinas de estados finitos obedecem a exatamente as mesmas propriedades TL, pode-se dizer que elas são *trace* equivalentes. Partindo dessa afirmação, para poder afirmar que duas máquinas de estados finitos não são *trace* equivalentes basta encontrar um propriedade TL que uma máquina de estados finitos satisfaça, que a outra não [BK+08].

Tendo duas máquinas de estados finitos, TS e TS', e a função $Traces(S)$, que representa todos os *traces* de uma máquina de estados finitos S, podemos então dizer:

Traces(TS) = *Traces(TS')* $\iff$ *TS* e *TS'* satisfazem as mesmas propriedades TL.

4.1.1 Exemplo

Utilizando uma adaptação do exemplo dos modelos para uma máquina de bebidas dada por Baier et al. [BK+08], mostrados na Figura 4.2, mostraremos como duas máquinas de estados finitos, embora aparentemente diferentes, podem ser *trace* equivalentes.

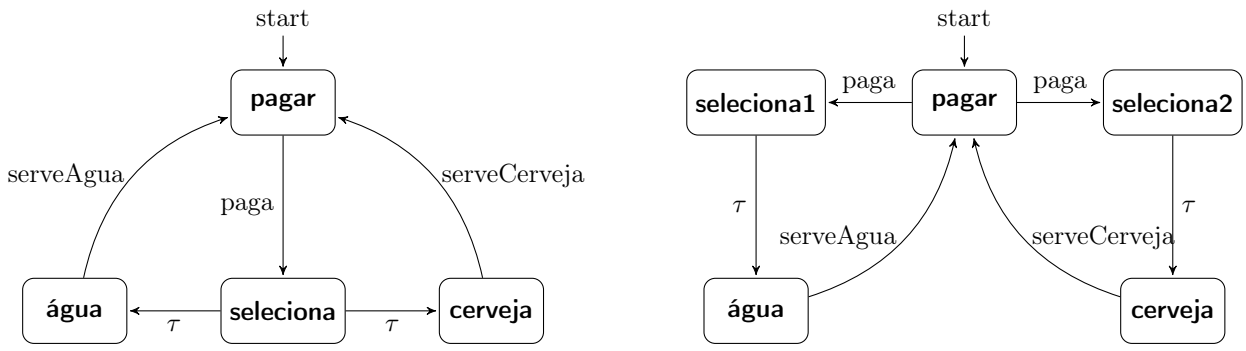


Figura 4.2: *Modelos para máquinas de bebidas, adaptada de [BK+08].*

Em ambos os modelos, podemos ver que após o pagamento existe a opção de selecionar a bebida, e em seguida a opção escolhida é servida. A diferença é que na primeira máquina de estados finitos só existe um estado de seleção, enquanto na segunda existem dois estados de seleção. Se analisarmos detalhadamente as máquinas de estados finitos, fica claro que embora com estruturas diferentes, ambas as máquinas de estados finitos só podem executar os mesmos *traces*, Tabela 4.1, e portanto são *trace* equivalentes.

Traces
((paga - τ - serveAgua) $\vee$ (paga - τ - serveCerveja))*

Tabela 4.1: *Traces possíveis nas máquinas de estados finitos da Figura 4.2*

Entretanto, se em nosso trabalho partimos da ideia de uma especificação *statechart* que define tudo que pode acontecer com o objeto, e que a implementação será um refinamento disso, significa que a implementação não irá necessariamente realizar todas as ações previstas na especificação. Sabendo disso queremos uma relação diferente da *trace equivalence*, pois não queremos verificar se os dois modelos são "iguais", e sim, se os *traces* das implementação, estão contidos nos *traces* da especificação, *Traces(I)* $\subseteq$ *Traces(E)*. Para estabelecer esse tipo de relação temos então o *Trace Inclusion* [BK+08].

4.2 *Trace Inclusion*

Trace Inclusion é uma relação de ordem parcial, que pode ser comparada com a *trace equivalence*. Estabelecer a relação *trace inclusion* entre duas máquinas de estados finitos *TS* e *TS'*, requer que todos os *traces* possíveis em *TS* existam em *TS'*, *Traces(TS)* $\subseteq$ *Traces(TS')*, porém diferente do *trace equivalence*, aqui a máquina de estados finitos *TS'* pode conter outros *traces* que não estejam em *TS*. Na Tabela 4.2 é possível ver o que distingue essas duas relações. A *trace inclusion* é geralmente associada com uma relação de implementação, onde ***Traces(TS)*** $\subseteq$ ***Traces(TS')*** **significa que *TS* é uma "implementação correta" de *TS'*** [BK+08].

Trace Equivalence	$Traces(TS) = Traces(TS')$
Trace Inclusion	$Traces(TS) \subseteq Traces(TS')$

Tabela 4.2: *Trace Equivalence e Trace Inclusion.*

Definição 3 *Sendo P uma propriedade TL, TS satisfaz P, representado por TS* $\models$ *P, se e somente se todos os traces de TS estão em P, Traces(TS)* $\subseteq$ *P [BK+08].*

Teorema 1 (*Trace Inclusion* e Propriedades de Tempo Linear) *Temos as máquinas de estados finitos TS e TS', com os mesmos conjuntos de proposições. Então as seguintes afirmações são equivalentes [BK+08]:*

1. *Traces(TS)* $\subseteq$ *Traces(TS')*
2. *Para qualquer Propriedades de Tempo Linear P: TS'* $\models$ *P implica em TS* $\models$ *P.*

Baier e Katoen et al. [BK+08] provam esse teorema da seguinte maneira:

1 $\Rightarrow$ *2 : Supondo que Traces(TS)* $\subseteq$ *Traces(TS'), e sendo P uma propriedades que TS' satisfaz, TS'* $\models$ *P. Pela Definição 3, sabemos que Traces(TS')* $\subseteq$ *P. Dado que Traces(TS)* $\subseteq$ *Traces(TS'),*

então Traces(TS) ⊆ P. Novamente pela Definição 3, podemos dizer que TS ⊨ P.

2 ⇒ 1 : Suponha que todas as propriedades TL serão consideradas: TS' ⊨ P implica que TS ⊨ P. Sendo P = Traces(TS'), podemos concluir que TS' ⊨ P, já que Traces(TS') ⊆ Traces(TS'). Assumindo que TS ⊨ P. Consequentemente Traces(TS) ⊆ Traces(TS').

Como dito no trabalho de Baier e Katoen et al. [BK+08], essa relação de *trace inclusion* é comumente utilizada para estabelecer relações de implementação e especificação, e por esse motivo optou-se por estabelecer esse tipo de relação entre os nossos modelos de implementação e especificação. A partir dessa relação de similaridade será possível verificar se todos os *traces* da implementação I, estão contidos na especificação E, *Traces(I)* ⊆ *Traces(E)*, e assim dizer se I é uma implementação correta de E. A seguir será explicado como as máquinas de estados finitos foram comparadas para saber se *Traces(I)* ⊆ *Traces(E)*

4.3 Simulação entre Especificação e Implementação

Uma maneira de verificar a relação de *trace inclusion* entre duas máquinas de estados finitos, é checar se existe uma relação de simulação entre elas [DHWT91]. Visto isso, decidimos então realizar uma simulação entre as máquinas de estados finitos, a fim de checar a relação de *trace inclusion* entre a implementação e a especificação.

Definição 4 *Uma máquina de estados finitos E simula uma máquina de estados finitos I se e somente se é possível passar em E, cada caminho que acontece em I, Traces(I) ⊆ Traces(E) [GL94].*

Realizando esta simulação, será verificado se os *traces* da implementação estão todos presentes na especificação, para isso a simulação irá percorrer os *traces* da implementação, e verificar se a especificação possui cada um deles. Caso exista algum *trace* que a implementação realiza, e a especificação não consegue realizar, então podemos dizer que existe um *trace* na implementação não previsto na especificação, ou seja *Traces(I)* ⊈ *Traces(E)*. Cada um desses *traces* não previstos serão considerados um Possível Ponto de Falha (PPF).

Para realizar essa simulação, temos que lembrar das transições τ que foram inseridas no modelo da implementação, Definição 1. Como queremos que essas transições sejam "ignoradas" no momento da simulação, definimos dois tipos de transições, como pode ser visto na Definição 5.

Definição 5 *Tipos de transição:*

- $q \overset{w}{\Rightarrow} q'$*: Representa um caminho do estado q para o estado q′ pelo evento w, podendo conter transições τ entre eles.*
- $q \xrightarrow{w} q'$*: Representa um caminho do estado q para o estado q′ pelo evento w.*

Para exemplificar, tome como modelos as máquinas de estados finitos da Figura 4.3. Pode-se dizer que na primeira máquina de estados finitos (esquerda) existe a transição $A1 \overset{w}{\Rightarrow} A2$, e na segunda (direita) existe a transição $B1 \xrightarrow{w} B2$.

Figura 4.3: *Exemplo de máquinas de estados finitos.*

Devido a inserção destas transições τ no modelo da implementação, foi necessário definir nossa própria relação de simulação. Nossa definição, Definição 6, foi baseada na ideia de *simulation order*, vista no trabalho de Baier et al. [BK+08].

Definição 6 *Sejam* $E = \{Q_E, \Sigma_E, \delta_E, qE_0\}$ *e* $I_\tau = \{Q_I, \Sigma_\tau, \delta_\tau, qI_0\}$ *duas máquinas de estados finitos que representem o modelo da especificação e da implementação respectivamente. Dizemos que a máquina* E *simula* I_τ, $E \preceq I_\tau$, *quando o estado inicial de* E *é capaz de simular o estado inicial de* I_τ, $qE_0 \preceq qI_0$. *Um estado* qE_0 *simula um estado* qI_0 *quando:*

- *Para toda transição* $qI_0 \overset{w}{\Rightarrow} qI'$, *existe uma transição* $qE_0 \overset{w}{\rightarrow} qE'$, *e* $qE' \preceq qI'$.

A partir dessa definição de simulação, desenvolvemos então a função S, Algoritmo 3, para realizar a simulação entre as máquinas de estados finitos. A função S recebe como parâmetro dois estados os quais ela utiliza para iniciar a simulação das máquinas de estados finitos. Nesta simulação a função verifica se para todo *trace* $qI \overset{w}{\Rightarrow} qI'$ possível na implementação, existe um *trace* $qE \overset{w}{\rightarrow} qE'$ na especificação. Se a especificação não conseguir simular, ou seja, se não houver um *trace* $qE \overset{w}{\rightarrow} qE'$, significa que a simulação falhou, e portanto um PPF foi identificado, caso contrário, o processo é feito novamente com os novos estados, $S(qI', qE')$. O processo irá se repetir até que todos os *traces* de I tenham sido percorridos. Se ao final da simulação a especificação conseguiu simular todos os *traces* da implementação, dizemos que a implementação é um implementação correta da especificação.

Definição 7 *Função* S *de simulação.*

- $S(Q_I, Q_E) \rightarrow \{boolean\}$

Algoritmo 3: Função S

```
1  S(qI, qE)
2  para ∀w ∈ Σ_qI faça
3      se ∃(qI ⇒^w qI') então
4          se ∃(qE →^w qE') então
5              retorna S(qI', qE')
6          fim
7          senão
8              retorna Falso
9          fim
10     fim
11 fim
12 retorna Verdadeiro
```

Para conseguir realizar essa comparação serão necessários dois modelos no formato de máquinas de estados finitos, seguindo a definição de máquina de estados finitos, dada na Seção 2.1.

- Q : Conjunto não vazio de estados;
- Σ : É um conjunto finito de chamadas de métodos, incluindo o τ no alfabeto da implementação;
- $\delta : Q \times \Sigma \rightarrow Q$ é a função de transição de estados;
- $q0 \in Q$ é o estado inicial;

Portanto, teremos o modelo E representando a especificação, que será a máquina de estados finitos obtida a partir da transformação do *statechart*, Seção 2.3, e o modelo I_τ representando a implementação já com a omissão dos métodos excessivos, Seção 3.3, seguindo a seguinte definição:

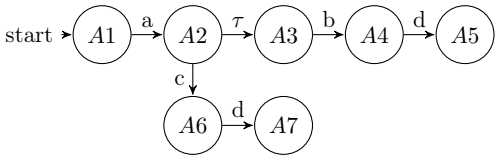
- E - Especificação : $\{Q_E, \Sigma_E, \delta_E, qE_0\}$;
- I_τ - Implementação : $\{Q_I, \Sigma_\tau, \delta_\tau, qI_0\}$;

Quando essas duas máquinas de estados finitos são simuladas, o objetivo é que essa simulação seja bem sucedida, Definição 8, e com isso garantir que tudo que está sendo realizado na máquina de estados finitos da implementação, esteja previsto na especificação, ou caso contrário, apontar qual *trace* não foi simulado.

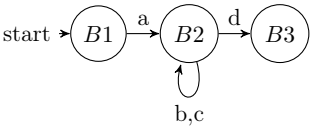
Definição 8 *Sejam $E = \{Q_E, \Sigma_E, \delta_E, qE_0\}$ e $I_\tau = \{Q_I, \Sigma_\tau, \delta_\tau, qI_0\}$ duas máquinas de estados finitos que representem o modelo da especificação e da implementação respectivamente. Uma simulação é bem sucedida quando $S(qI_0, qE_0) = True$.*

4.4 Exemplo

Para ilustrar a relação estabelecida neste trabalho, suponha que a máquina de estados finitos TS, Figura 4.4a, foi a máquina de estados finitos obtida a partir da implementação do software, e que a máquina de estados finitos TS', Figura 4.4b, representa a máquina de estados finitos da especificação. Dadas essas duas máquinas de estados finitos, o objetivo agora é saber se $Traces(TS) \subseteq Traces(TS')$. Utilizando a função S dada no Algoritmo 3, iremos verificar a nossa relação de simulação entre as máquinas de estados finitos, iniciando a simulação a partir de seus estados inicias, $S(A1, B1)$.



(a) *TS - Implementação.*



(b) *TS' - Especificação.*

Figura 4.4: *Exemplo de máquinas de estados finitos de Implementação e Especificação.*

Os passos executados pela função S durante essa simulação, mostrados na Tabela 4.3, são os seguintes:

	$S(qI, qE)$	**Implementação**	**Especificação**
1	$S(A1, B1)$	$A1 \stackrel{a}{\Rightarrow} A2$	$B1 \stackrel{a}{\rightarrow} B2$
2	$S(A2, B2)$	$A2 \stackrel{b}{\Rightarrow} A4$	$B2 \stackrel{b}{\rightarrow} B2$
		$A2 \stackrel{c}{\Rightarrow} A6$	$B2 \stackrel{c}{\rightarrow} B2$
3	$S(A4, B2)$	$A4 \stackrel{d}{\Rightarrow} A5$	$B2 \stackrel{d}{\rightarrow} B3$
4	$S(A6, B2)$	$A6 \stackrel{d}{\Rightarrow} A7$	$B2 \stackrel{d}{\rightarrow} B3$
5	$S(A5, B3)$	-	-
6	$S(A7, B3)$	-	-

Tabela 4.3: *Traces simulados.*

- Passo 1: A única ação possível no estado da implementação, $A1$, é a ação a, que leva ao estado $A2$. A especificação por sua vez consegue também realizar a mesma ação indo para o estado $B2$. O passo seguinte então será a simulação dos estados resultantes, $S(A2, B2)$;

- Passo 2: No estado $A2$ existem dois caminhos possíveis, o primeiro pelo evento b que leva ao estado $A4$, e pelo evento c que conduz para o estado $A6$. Em ambos os casos, o modelo da especificação consegue realizar a mesma ação permanecendo no mesmo estado, $B2$, e portanto os passos seguintes consistem em analisar $S(A4, B2)$ e $S(A6, B2)$;

- Passo 3: Neste passo, é realizada a simulação $S(A4, B2)$, onde a única ação permitida é a ação d, que conduz ao estado $A5$ da implementação, e que conduz ao estado $B3$ no modelo da especificação;

- Passo 4: Aqui a simulação realizada é $S(A6, B2)$, onde somente o evento d é permitido. Na implementação essa ação conduz ao estado $A7$, e na especificação, ao estado $B3$.

- Passos 5 e 6 : Ao chegar nos estados $A5$ e $A7$ a implementação não possibilita mais nenhuma ação, e portanto esses *traces* chegaram ao seu final. Como até o momento a especificação conseguiu simular todas as ações da implementação, dizemos que os *traces* foram reconhecidos na especificação e a simulação foi bem sucedida.

Com essa simulação finalizada, é possível afirmar que TS' conseguiu simular TS, e portanto $Traces(TS) \subseteq Traces(TS')$. Com isso sabemos que TS é uma correta implementação de TS'.

4.5 Conclusão

Para entender melhor como funciona essa comparação, podemos imaginar um jogo de imitação, no qual qualquer ação que o modelo da implementação fizer, o modelo da especificação deve ser capaz de fazer também. Fazemos esse jogo de imitação até ter percorrido todos os *traces* de I, e se em algum momento o modelo E tentou imitar I e não conseguiu realizar a ação, significa que E não conseguiu simular I, e que esse *trace* é um Possível Ponto de Falha na implementação. Os Possíveis Pontos de Falhas não são necessariamente erros no software, e sim diferenças entre implementação e especificação. Essas diferenças podem ser causadas por implementações erradas, incompletudes na especificação, métodos com nomes diferentes, entre outras razões, e por isso é necessário a intervenção humana no processo, pois quando a função S retorna um PPF, cabe ao desenvolvedor analisar a implementação e a especificação, e definir o porquê dessa divergência entre os modelos estar acontecendo.

Através dessa comparação é possível ter uma garantia de que todos os *traces* que estão sendo permitidos na implementação do software já estão previstos na especificação, contudo não há garantias de que todos os *traces* definidos na especificação estejam implementados no software. Outra ponto a ficar atento é de que essa abordagem está fortemente relacionada a qualidade da especificação, visto que ela deve descrever **todos** os *traces* permitidos, além de que os métodos que não forem descritos na especificação serão omitidos na implementação, ou seja, quanto mais métodos a especificação descreve, mais preciso será o modelo da implementação.

No próximo capítulo serão mostrados os estudos de caso realizados, onde comparamos alguns softwares com suas respectivas especificações *statecharts*. Os estudos de caso onde não houve PPFs apontados nos garantiram uma correta implementação por parte do software. Já nos casos onde houve PPFs apontados, houve uma análise manual das implementações e das especificações para definir os motivos que causaram esses PPFs.

Capítulo 5

Prova de Conceito

Neste capítulo, apresentaremos como aplicar nossa abordagem, assim como estudos de caso como prova de conceito da aplicabilidade de nossa metodologia, proposta nos capítulos anteriores. A seguir veremos os passos para a aplicação da abordagem proposta.

5.1 Passos para a Aplicação da Abordagem

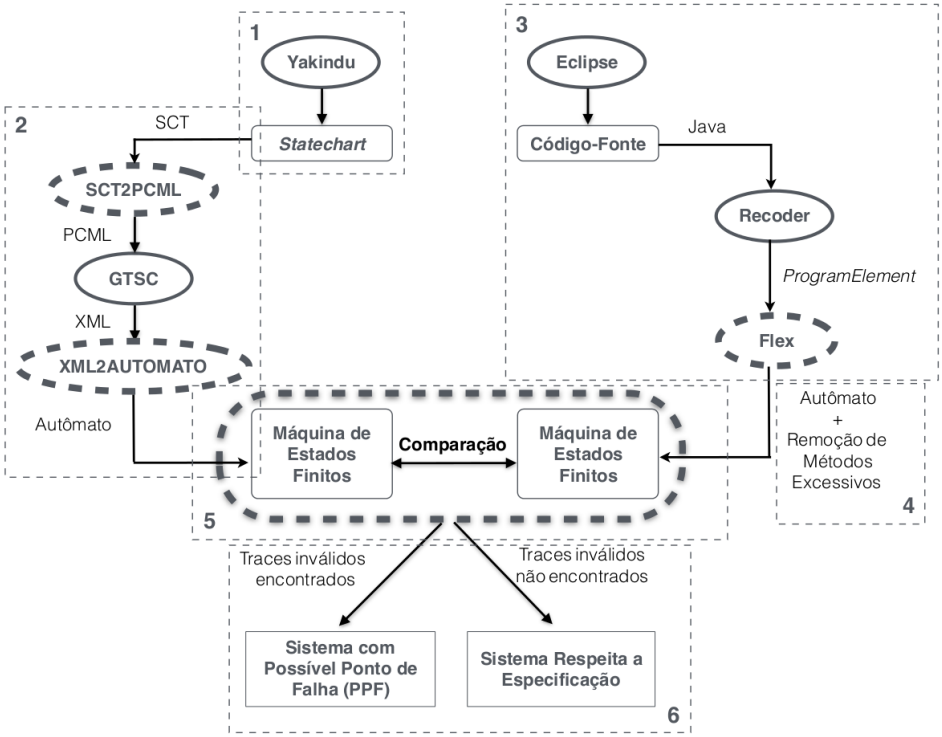


Figura 5.1: *Passos da aplicação da abordagem.*

Para realizar os estudos de caso, primeiramente selecionamos os softwares, com suas respectivas especificações escritas em *statecharts*, e depois aplicamos nossa abordagem, que está ilustrada na Figura 5.1, seguindo os seguintes passos:

1. **Modelagem da Especificação**: O primeiro passo é modelar o *statechart* da especificação na ferramenta Yakindu [yak];

2. **Conversão da Especificação**: Em seguida é necessário converter o *statechart* da especificação, definido no passo anterior, em uma máquina de estados finitos, Seção 2.3. Para isso ele passa por três conversores, sendo eles:

- SCT2PCML;
- GTSC;
- XML2AUTOMATO.

3. **Mineração do Modelo da Implementação**: Com o modelo da especificação já no formato de uma máquina de estados finitos, o próximo passo é obter o modelo que represente o comportamento do software, Seção 3.2. Nesta tarefa o `Recoder` [rec] analisa e desmembra todas as estruturas contidas no código-fonte Java, e com essas informações o algoritmo Flex monta a máquina de estados finitos que representa o possível comportamento do objeto no software;

4. **Remoção dos Métodos Excessivos**: Para permitir uma comparação entres as máquinas de estados finitos, decidimos deixar ambas com o mesmo alfabeto de métodos, e por isso neste passo iremos remover os métodos que existem na implementação, mas que não estão descritos na especificação, Seção 3.3;

5. **Comparação das máquinas de estados finitos**: O passo seguinte é comparar as duas máquinas de estados finitos, e averiguar se existe algum *trace* possível na implementação, que não esteja previsto na especificação, Seção 4.3;

6. **Análise dos *Traces* Encontrados**: Caso a comparação retorne algum *trace*, esse *trace* é apontado como um PPF. Nesta etapa é feita uma análise manual do código-fonte e da especificação, a fim de identificar a razão dos *traces* apontados estarem sendo permitidos, além de verificar se estes PPFs realmente irão implicar em erros no software.

Esses passos foram seguidos em todos os estudos de casos realizados neste trabalho, os quais foram:

- **Socket:** Esse estudo de caso nos permitiu verificar se alguns sistemas selecionados estavam utilizando a classe `java.net.Socket` de acordo com a sua especificação.

- **ATM:** O sistema de uma máquinas ATM (*Automated Teller Machine*) se mostrou um bom exemplo para nossa abordagem por possuir diversas especificações e implementações disponíveis online. Este estudo de caso nos possibilitou checar se uma simulação em uma ATM está respeitando a especificação da mesma.

- **Máquina de Café:** Neste estudo de caso, nós implementamos um código java, baseado em nossa interpretação do *statechart* da especificação, e depois, checamos se nossa interpretação da especificação estava de correta.

- **SWPDC:** Esse estudo de caso nos propiciou a utilização da nossa abordagem em cenário real, visto que analisamos o software SWPDC, software esse desenvolvido e utilizado pelo INPE (Instituto Nacional de Pesquisas Espaciais).

Para realizar a mineração do comportamento do software utilizamos o método `Flex.generate` que nós mesmo implementamos. Os parâmetros necessários para esse método são:

- **Cenário:** Local do software onde há uma instância do objeto a ser observado. Neste parâmetro criamos uma instância da classe `Klass`, onde passamos como parâmetros no construtor o nome da classe a ser observada, o nome do método a ser observado, e os parâmetros deste método (para o caso de sobrecarga de métodos);

- **O objeto a ser observado:** Objeto o qual queremos saber o possível comportamento quando o software for executado. Aqui é passado o nome do objeto a ser observado dentro do cenário passado no primeiro parâmetro.

Todos nossos estudos de caso foram executados dentro do software Eclipse IDE, na versão Luna Release (4.4.0) [ecl], rodando o Java em sua versão 7. O hardware utilizado foi um notebook Dell Vostro 5480, com processador Intel Core i7 2.40 GHz e 8GB de memória RAM.

5.2 Estudo de Caso com a Classe Socket

A classe `Socket` implementa soquetes, que são pontos finais de uma comunicação entre duas máquinas, o que a torna bastante utilizada para a troca de informações pela rede. Para utilizar um `Socket` é necessário antes de tudo estabelecer uma conexão entre as máquinas, depois o `Socket` pode receber e enviar informações, e ao final, a conexão deve ser fechada. Neste primeiro estudo de caso, verificamos em 5 sistemas de código aberto se a classe `java.net.Socket` está sendo usada de acordo com o comportamento definido em sua especificação.

5.2.1 Definição dos Modelos

O primeiro estudo de caso foi iniciado com a construção de um modelo da classe `java.net.Socket` baseado no padrão *statechart* e nas especificações da classe `java.net.Socket` encontradas em [FYD+08]. Este modelo da especificação será chamado de Máquina E, e pode ser visto na Figura 5.2.

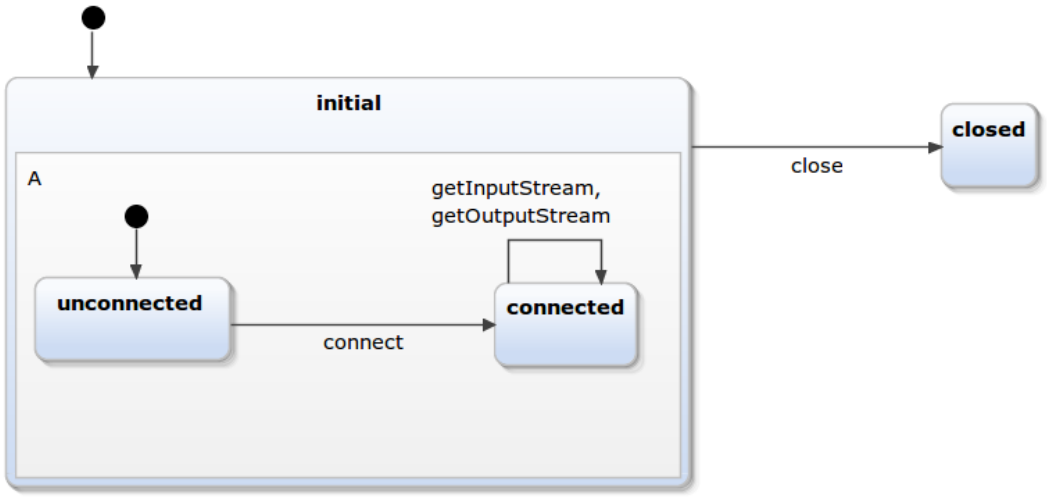


Figura 5.2: *Especificação da classe `java.net.Socket`.*

Concluída a modelagem da classe na forma de um diagrama *statechart*, passamos à transformação do mesmo diagrama em uma máquina de estados finitos, como descrito na Seção 2.3.2. O resultado da dessa conversão pode ser visto na Figura 5.3.

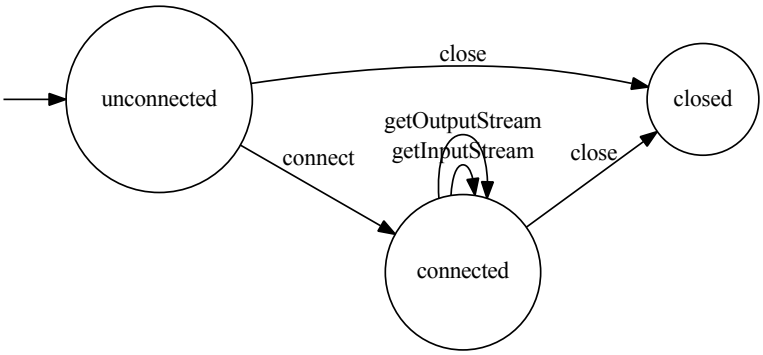


Figura 5.3: *Statechart E no formato de uma máquina de estados finitos.*

Um vez construída a máquina E, como descrito acima, devemos obter os modelos da implementação. Para este estudo selecionamos um conjunto softwares escritos em Java, com códigos-fonte disponíveis e que fizessem uso da classe `java.net.Socket`. Identificamos manualmente os locais onde a classe `Socket` era usada, para dessa forma indicar ao algoritmo Flex qual parte do software

observar, e gerar os modelos. Como houve casos onde o `Socket` somente era instanciado, e não utilizado, optamos por analisar somente os sockets que realizassem ao menos duas chamadas de métodos no sistema.

Nesse estudo de caso analisamos cinco softwares de código aberto que fazem uso da classe `java.net.Socket`, identificados na Tabela 5.1. Usaremos o software *yafta* para ilustrar como fizemos a análise para todos os softwares apresentados na tabela. Na Listagem 5.4 temos a classe definida para observar as ocorrências do `Socket` dentro do software *yafta*. Neste software temos duas classes que manipulam um objeto do tipo `Socket`, sendo elas a *UserProtocolInterpreter* e a *DataConnection*. No código podemos ver o método `Flex.generate`, que recebe como parâmetros: a classe, o método e a lista de parâmetros desse método (para o caso de sobrecarga de métodos) que serão o cenário a ser observado, além de receber também o objeto a ser observado no cenário. As informações sobre as máquinas de estados finitos geradas pelo algoritmo Flex, para todos softwares que observamos neste estudo de caso podem ser vistas na Tabela 5.1, onde cada coluna representa:

- **Software:** Nome do software observado;
- **Estados:** Soma do número de estados das máquinas de estados finitos obtidas de todos os cenários do software;
- **Transições:** Soma do número de transições das máquinas de estados finitos obtidas de todos os cenários do software;
- **Cenários:** Quantidade de locais no software onde há uma instância do objeto observado;
- **Tempo de Geração (ms):** Soma do tempo (em milissegundos) que o algoritmo Flex levou para gerar as máquinas de estados finitos de cada cenário do software.

	Software	Estados	Transições	Cenários	Tempo de Geração (ms)
1	yafta	8	6	2	532
2	ganymed SSH-2	41	43	6	981
3	evosuite 1.0.2	41	38	7	1524
4	aamfetch 0.9.1	12	18	1	165
5	lucene 6.0.0	13	15	2	380

Tabela 5.1: *Dados da mineração das máquinas de estados finitos dos sistemas para a classe* `Socket`.

Durante a realização deste estudo de caso percebeu-se que a classe `Socket` permite estabelecer conexões de diferentes maneiras, não só através do método `connect`, como descrito na especificação, e por essa razão foi necessário definir que essas outras formas de estabelecer conexão, Tabela 5.2, serão equivalentes ao método `connect` que está descrito na especificação, pois caso contrário, essas ações seriam omitidas no modelo da implementação, por não estarem descritas na especificação, como foi dito na Seção 3.3. Definimos então que a instanciação de um objeto s da classe `java.net.Socket` é equivalente ao método `connect()`, e que uma atribuição direta do método `accept()` de uma instância da classe `java.net.ServerSocket` a um objeto instância da classe `java.net.Socket`, também corresponde ao método `connect()` da especificação, Tabela 5.2.

O passo seguinte é identificar as transições τ no modelo da implementação e omitir os métodos que não existam na especificação, Definição 1.

Gerados então os modelos, realizamos a comparação entre eles. Para tanto, comparamos separadamente cada modelo obtido a partir da implementação com a especificação, e depois verificamos os PPFs apontados na comparação.

```
1  public class Observer{
2
3      public Automato exemplo1() {
4          this.classFullName = "yafta.ftp.UserProtocolInterpreter";
5          return Flex.generate(new Klass(classFullName, "connect", "String",
6                  "int"), "Socket");
7      }
8
9      public Automato exemplo2() {
10         this.classFullName = "yafta.ftp.DataConnection";
11         return Flex.generate(new Klass(classFullName, "connect"),
12                 "Socket");
13     }
14
15 }
```

Figura 5.4: *Definição dos pontos a serem observados no software yafta.*

Implementação	Especificação
Socket s = x.accept()	s.connect()
Socket s = new Socket(targetAddress, targetPort)	s.connect()

Tabela 5.2: *Ações equivalentes ao método `connect` da classe `java.net.Socket`.*

5.2.2 Análise de Similaridade

Realizamos a comparação individualmente de cada um dos modelos obtidos a partir dos softwares com a especificação, para verificar se as implementações estavam de acordo com suas especificações. Dado os PPFs apontados após a comparação, verificamos manualmente o código e a especificação, para entender o porquê destes *traces* terem sido apontados como possíveis pontos de falhas. Na Tabela 5.3 temos as seguintes informações sobre as verificações realizadas.

- **Software:** Nome do software observado;
- **Cenários:** Quantidade de locais no software onde há uma instância do objeto observado;
- **Alfabeto:** Soma da quantidade de métodos que o objeto observado executa;
- **Tempo de Simulação (ms):** Soma do tempo (em milissegundos) que demorou para executar a simulação das máquinas de estados finitos;
- **PPF:** Quantidade de *traces* inválidos encontrados no software.

	Software	Cenários	Alfabeto	Tempo de Simulação(ms)	PPF
1	yafta	2	4	35	0
2	ganymed	6	11	122	4
3	evosuite	7	23	100	2
4	aamfetch	1	5	7	1
5	lucene	2	8	28	0

Tabela 5.3: *Resultados da simulação com classe `Socket`.*

Como já foi dito, os Possíveis Pontos de Falhas (PPF) não são necessariamente erros, e sim pontos que necessitam de uma maior atenção por parte dos desenvolvedores. Com base neste princípio, foi realizada a avaliação manual do código-fonte para entender o porquê dos *traces* apontados terem sido permitidos. Os sistemas Yafta e Lucene não apontaram nenhum PPF, já os outros três retornaram um total de 7 PPFs. A seguir será explicado o que fez com que esses *traces* fossem apontados pela comparação.

5.2.2.1 Ganymed

Ganymed é uma biblioteca Java que implementa o protocolo SSH-2. Nesta biblioteca foram identificados 4 PPFs. Após analisar as classes do sistema, foi descoberto que os PPFs foram apontados pelos seguintes motivos.

- Erro 1 - Um máquina de estados finitos minerada da classe `LocalAcceptThread` possibilitava a chamada de um método `getOutputStream` após o método `close`, sequência essa que não é permitida na especificação. Após analisar o código, percebemos que o método `getOutputStream` estava dentro de um bloco `try-catch` que tratava a exceção `IOException`, exceção do qual estende a `java.net.SocketException`, que é a exceção lançada quando é tentado executar o método `getOutputStream` com o `socket` fechado, ou seja, o próprio sistema já estava preparado caso esse *trace* ocorresse.

- Erros 2 e 3 - Em dois locais do software, classes `RemoteAcceptTread` e `RemoteX11Accept`, foram apontados que seria permitido chamar o método `close` logo após a chamada de um outro `close`. Ambos PPFs foram apontados por uma limitação da nossa abordagem na mineração do código. Esses métodos `close` se encontravam, o primeiro num bloco `try`, e o segundo no `catch`. Como nossa mineração não considera os blocos `try-catch`, ele considerou que ambos poderiam acontecer, quando na verdade com o sistema em execução, somente um iria ser executado.

- Erro 4 - Na classe `TransportManager` foi identificado que poderia haver duas tentativas de conexão seguidas, método `connect`, o que não é permitido na especificação. Analisando o código foi notado que cada `connect` se encontrava dentro de um bloco `if`, `if(proxyData == null)` e `if(proxyData instanceof HTTPProxyData)`, ou seja, o sistema executaria somente o primeiro `if`, ou somente o segundo, mas nunca os dois. Nossa mineração é estática, e portanto não consegue analisar o resultados das operações lógicas, e por isso foi considerado que ambos os `ifs` poderiam ser executados, o que acarretaria na execução dos dois `connects`.

5.2.2.2 Evosuite

EvoSuite é uma ferramenta que gera automaticamente casos de teste para classes escritas em código Java. Neste estudo de caso, foram apontados 2 PPFs. Após a análise manual do código-fonte do sistema, foi identificado que os PPFs são permitidos pelo seguinte motivo:

- Erros 1 e 2 - Nas classes `LoggingUtils` e `SpawnProcessKeepAliveCheckerIntelliJ`, o comando `socket = serverSocket.accept()`, que é equivalente ao método `connect` da especificação, pode ser executado duas ou mais vezes seguidas. Isso ocorre porque este comando se encontra dentro de um bloco `while`, e por esta razão na primeira iteração o comando será executado, e na próxima iteração irá executá-lo novamente. Neste caso, após analisar manualmente o código-fonte, percebeu-se que estabelecer uma nova conexão sobre uma outra já estabelecida, é uma sequência que pode ser permitida, e não irá lançar nenhuma exceção no software, mesmo que esse *trace* não esteja descrito na especificação.

5.2.2.3 Aamfetch

AamFetch é um utilitário para buscar mensagens do grupo de notícias alt.anonymous.messages (ou outros grupos). No software aamfetch, foi identificado 1 PPF. Após a análise do código-fonte do sistema, foi identificado que este PPF ocorreu pelo seguinte motivo:

- Erro 1 - Na classe `Fetcher` foi apontado que seria permitido chamar o método `close` logo após a chamada de um outro `close`, como ocorreu nos erros 2 e 3 no software Ganymed. Neste caso, a primeira chamado do método `close` está dentro de um `if` e portanto este pode ou não ser executado, enquanto o segundo se encontra ao final do método. Após a análise, ficou entendido que não haveria problema em executar um `close` com o `socket` já fechado.

Dentre os estudos de caso realizados com a classe `java.net.Socket`, foram identificados 7 PPFs, Tabela 5.3, entretanto nenhum deles realmente iria acarretar em uma exceção ou falha no software, mesmo que esses *traces* não estejam previstos na especificação. Após um análise do código-fonte e da especificação, percebeu-se que neste caso da classe `Socket`, muitos dos PPFs se deram pela especificação ser pobre, e não descrever de maneira completa as ações possíveis para esta classe. Sabendo disso, fica claro que faz-se necessário um *statechart* o mais detalhado possível, para reduzir o números de PPFs falsos positivos, que são os PPFs apontados que não geram falhas ou exceções na execução do software.

5.3 Estudo de Caso com uma ATM

Sistemas de máquinas ATM (*Automated Teller Machine*), caixa eletrônico, são sistemas onde o usuário insere seu cartão e, a partir disso, pode realizar várias transações financeiras. Este tipo de sistema é bastante utilizado como um sistema a ser verificado, visto que em teoria deve ser um sistema que não possui falhas, pois isso poderia causar perdas financeiras, e por possuir diversos modelos de especificação e exemplos de implementações disponíveis online.

Na busca por exemplos para validarmos nossa abordagem, encontramos no site da Gordon College [gor] um exemplo feito para simular uma ATM. Para realizar essa simulação eles fizeram especificações, entre elas *statecharts*, e também foi feita uma implementação em Java do sistema. Sabendo então que existem as especificações *statecharts* e a implementação Java, esse sistema estava habilitado a ser usado como estudo de caso neste trabalho.

5.3.1 Definição dos Modelos

Para utilizar esse sistema ATM, adaptamos o *statechart* disponível, e fizemos as seguintes modificações:

- um mapeamento entre sistema e especificação para relacionar os nomes dos métodos do código-fonte com as ações do *statechart*, e a
- criação de um estado hierárquico para reduzir o número de transições no modelo.

Na Figura 5.5 pode ser visto o modelo *statechart* da especificação após as modificações feitas.

Com a modelagem da especificação feita, realizamos então o processo de transformação da mesma, a fim de obtermos a máquina de estados finitos dessa especificação. A máquina de estados finitos obtida a partir da especificação dada na Figura 5.5 pode ser vista na Figura 5.6.

Uma vez que a especificação se encontra no formato de uma máquina de estados finitos, passamos agora para o modelo da implementação. O sistema foi implementado com uma interface gráfica, com menus contendo as operações que podem ser executadas, ou seja o usuário escolhe de

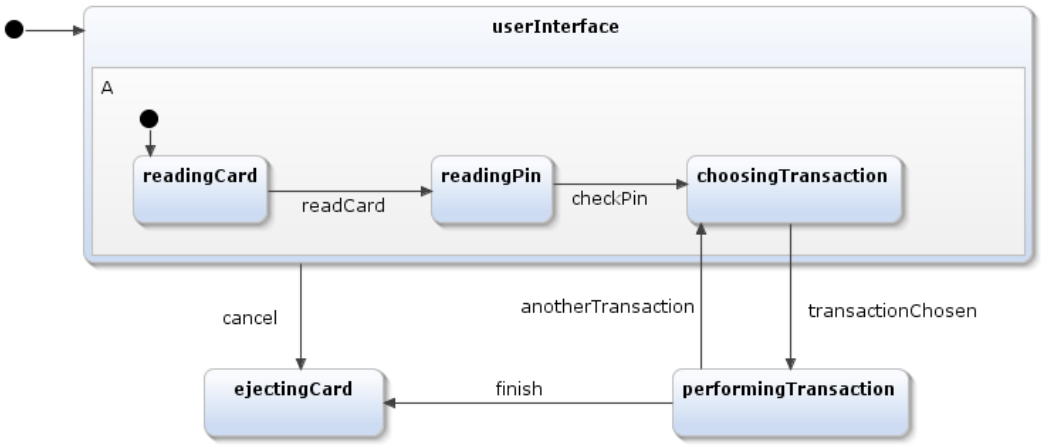


Figura 5.5: *Especificação de uma ATM.*

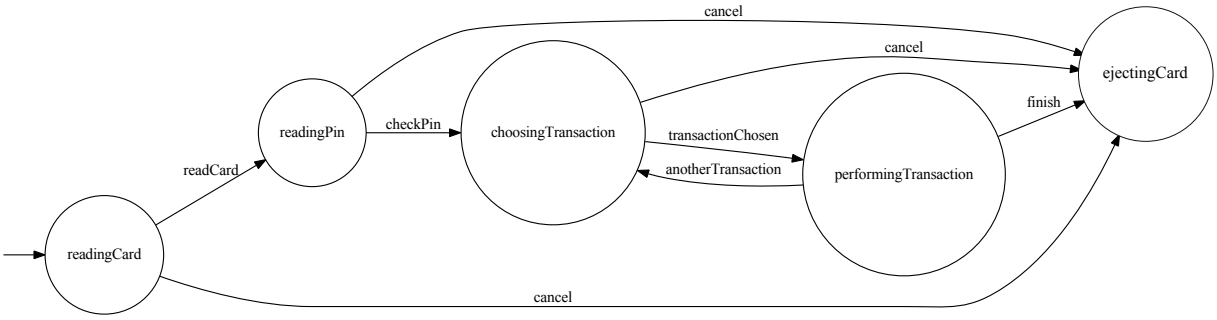


Figura 5.6: *Especificação de uma ATM no formato de uma máquina de estados finitos.*

maneira aleatória as ações que irão acontecer no sistema. Neste trabalho é realizada uma análise do código-fonte, ou seja, sem o sistema estar em execução, o que implica que o usuário não tem participação, e por isso não daria para extrair um modelo que representasse o comportamento desse software. Entretanto nesse mesmo sistema existe uma classe `SimulatedBank.java`, Listagem 5.1, que realiza uma simulação da ATM implementada, sem qualquer interação com o usuário, através do método `handleMessage`. Decidimos então analisar essa classe de simulação, e compará-la com a especificação, de forma a garantir que a simulação obedeça o *statechart* da especificação.

```
1  package simulation;
2  import banking.AccountInformation;
3  import banking.Balances;
4  import banking.Card;
5  import banking.Message;
6  import banking.Money;
7  import banking.Status;
8
9  public class SimulatedBank
10 {
11     int cardNumber;
12
13     public Status handleMessage(Message message, Balances balances) {
14         Status result;
15         SimulatedBank bank = new SimulatedBank();
16
17         cardNumber = bank.readCard(message);
18
19         result = bank.checkCard();
```

```
20
21          if(!(result instanceof Failure)){
22              result = bank.checkPin(message);
23              if(!(result instanceof InvalidPIN)){
24                  result = bank.transactionChosen(message, balances);
25              }else{
26                  return result;
27              }
28          }else{
29              return result;
30          }
31          return null;
32      }
33
34      ...
35  }
```

Listagem 5.1: *Classe SimulateBank.java.*

A classe `ObserverATM.java`, Listagem 5.2, foi feita para minerar o comportamento que o objeto instância da classe `SimulatedBank` irá realizar. Os dados da mineração podem ser vistos na Tabela 5.4.

```
1  import br.usp.classes.Klass;
2
3  public class ObserverATM{
4      public void examplo1(){
5          this.classFullName = "simulation.SimulatedBank";
6          return Flex.generate(new Klass(classFullName, "handleMessage", "
               Message", "Balances"),"SimulatedBank");
7      }
8  }
```

Listagem 5.2: *Classe ObserverATM.*

Software	Estados	Transições	Alfabeto	Tempo de Geração(ms)
Atm	7	8	6	574

Tabela 5.4: *Dados da mineração da máquina de estados finitos do sistema ATM.*

5.3.2 Análise de Similaridade

Com os dois modelos já montados, é hora de compará-los e checar se a implementação está de acordo com a especificação. Neste estudo de caso não foi identificado nenhum PPF, Tabela 5.5, ou seja as ações realizadas pelo objeto instância da `SimulatedBank` estão totalmente de acordo com a sua especificação *statechart*.

Software	Alfabeto	Tempo de Simulação (ms)	PPF
Atm	6	14	0

Tabela 5.5: *Dados da comparação das máquinas de estados finitos do sistema ATM.*

5.4 Estudo de Caso com uma Máquina de Café

Nesse estudo de caso decidimos verificar uma implementação feita por nós, onde dada uma especificação já montada, implementamos um sistema a partir da interpretação dessa especificação.

Com isso, foi possível ver se nossa interpretação da especificação estava realmente correta. Para realizar esse estudo decidimos utilizar um sistema bem simples de uma Máquina de Café, onde um usuário coloca uma moeda na máquina, escolhe a bebida e a máquina serve a opção escolhida.

5.4.1 Definição dos Modelos

Para realizar esse estudo de caso usamos o *statechart* encontrado no site [sta], o qual possui vários exemplos de especificações *statecharts*, entre eles uma representação simples do comportamento de uma Máquina de Café. Neste estudo tivemos que realizar um mapeamento dos nomes dos métodos do código com as ações da especificação. O *statechart* da especificação pode ser visto na Figura 5.7.

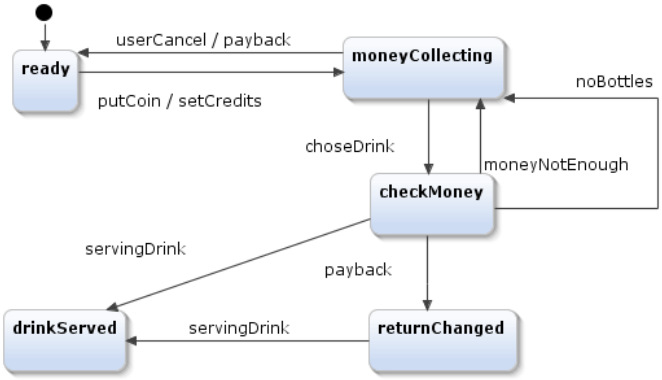


Figura 5.7: *Especificação do software de uma máquina de café.*

Como pode ser visto, o *statechart* usado como especificação já é quase uma máquina de estados finitos, e por isso a única modificação feita no *statechart* é a omissão dos eventos internos. A máquina de estados finitos obtida ao final das modificações pode ser vista na Figura 5.8.

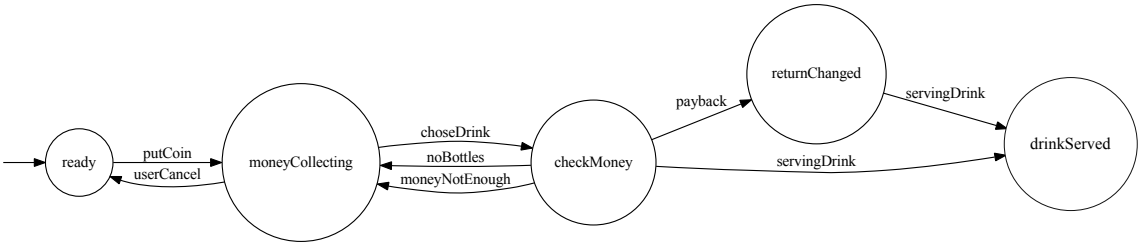


Figura 5.8: *Especificação do software de uma máquina de café no formato de uma máquina de estados finitos.*

Como modelos para a implementação, foram utilizados dois exemplos. O primeiro baseado no código bem simples de Simone Hanazumi [Han15], onde o adaptamos para separar os comandos em métodos diferentes, porém mantendo a mesma lógica. A classe `CoffeeMachineSimone.java` adaptada pode ser vista na Listagem 5.3. O segundo exemplo foi desenvolvido por nós, onde implementamos nossa interpretação da especificação *statechart* disponibilizada, e pode ser visto na Listagem 5.4.

```
1 package coffeeMachineExample;
2 import gov.nasa.jpf.jvm.Verify;
3 import java.io.IOException;
4 public class CoffeeMachineSimone {
```

```
5
6       private static CoffeeMachineSimone cm = null;
7
8       public static void main(String[] args) throws IOException {
9           cm = getInstance();
10          cm.putCoin();
11          cm.servingDrink(Verify.random(3));
12      }
13
14     ...
15  }
```

Listagem 5.3: *Classe da Máquina de Café usada no exemplo da Simone.*

```
1  package coffeeMachineExample;
2  import gov.nasa.jpf.jvm.Verify;
3  import java.io.IOException;
4  public class CoffeeMachineLuciano {
5
6       private static CoffeeMachineLuciano cm = null;
7
8       public static void main(String[] args) throws IOException {
9           cm = getInstance();
10          float coin = cm.putCoin();
11          int drink = cm.choseDrink();
12          if(coin >= priceOfDrink(drink)){
13              if(coin > priceOfDrink(drink)){
14                  cm.payback();
15              }
16              cm.servingDrink(drink);
17          }else{
18              cm.moneyNotEnough();
19          }
20      }
21
22      ...
23  }
```

Listagem 5.4: *Classe da Máquina de Café que implementamos.*

A classe `ObserverCoffeeMachine.java`, Listagem 5.5, foi feita para observar o comportamento das duas implementações. Os dados da mineração realizada podem ser vistos na Tabela 5.6.

```
1  import br.usp.classes.Klass;
2
3  public class ObserverCoffeeMachine{
4
5       public void examplo1(){
6           this.classFullName = "coffeeMachineExample.CoffeeMachineSimone";
7           return Flex.generate(new Klass(classFullName, "main", "String[]"),"cm"
                );
8       }
9
10      public void examplo2(){
11          this.classFullName = "coffeeMachineExample.CoffeeMachineLuciano";
12          return Flex.generate(new Klass(classFullName, "main", "String[]"),"cm"
                );
13      }
14  }
```

Listagem 5.5: *Classe ObserverCoffeeMachine.java.*

Sistema	Estados	Transições	Alfabeto	Tempo de Geração (ms)
CoffeeMachineSimone	4	3	3	281
CoffeeMachineLuciano	7	8	7	296

Tabela 5.6: *Dados da mineração das máquinas de estados finitos dos sistemas das Máquinas de Café.*

5.4.2 Análise de Similaridade

Com os dois modelos já montados, é hora de compará-los e checar se as implementações estão respeitando a especificação. No modelo da implementação de Simone, identificamos um PPF, Tabela 5.7, pois não havia a verificação da quantidade de moedas na Máquina de Café, o que permitia o *trace* $q \xrightarrow{putCoin} q', q' \xrightarrow{servingDrink} q''$, que embora não acarretasse em uma exceção no sistema, não é permitido na especificação. Na nossa implementação não identificamos nenhum PPF, o que nos permite dizer que a implementação que fizemos está correta, pois obedece o *statechart* da especificação. É importante ressaltar que a implementação de Simone é bem simples e foi feita com outro propósito, e por isso o PPF encontrado.

Sistema	Alfabeto	Tempo de Simulação (ms)	PPF
CoffeeMachineSimone	3	2	1
CoffeeMachineLuciano	7	7	0

Tabela 5.7: *Dados da comparação das máquina de estados finitos das Máquinas de Café.*

5.5 Estudo de Caso com o Software SWPDC - 1

Nesse Estudo de caso, iremos trabalhar com o software SWPDC (Software Piloto Embarcado do Payload Data Handling Computer) [Ach13], software esse que foi desenvolvido pelo INPE no contexto do projeto QSEE (Qualidade de Software Embarcado em Aplicações Espaciais) e possui uma implementação em Java, além de especificações escritas em *statecharts*.

5.5.1 Definição dos Modelos

O SWPDC conta com vários documentos de especificação, dentre os quais temos diagramas de classe, diagramas de atividades e *statecharts*. Em meio aos modelos de *statecharts* disponibilizados, utilizaremos o modelo definido pela colaboradora Ludmila Ferreira para gerenciar os modos de operação da aplicação, como modelo de especificação para esse estudo de caso. O *statechart* pode ser visto na Figura 5.9. A máquina de estados finitos gerada a partir desse *statechart* pode ser visualizada na Figura 5.10

Para a geração do modelo da implementação, observamos o objeto `fsm` contido no método `main` da classe `SWPDC.java`, Listagem 5.6, visto que este objeto é o responsável por gerenciar os estados da máquina. A seguir podemos ver o código-fonte da classe `SWPDC.java`, Listagem 5.6, e também da classe `FSM.java`, Listagem 5.7.

```
1 package swpdc;
2
3 import dados.GerenciadorDados;
4 import dados.Housekeeper;
5 import dados.Temperatura;
```

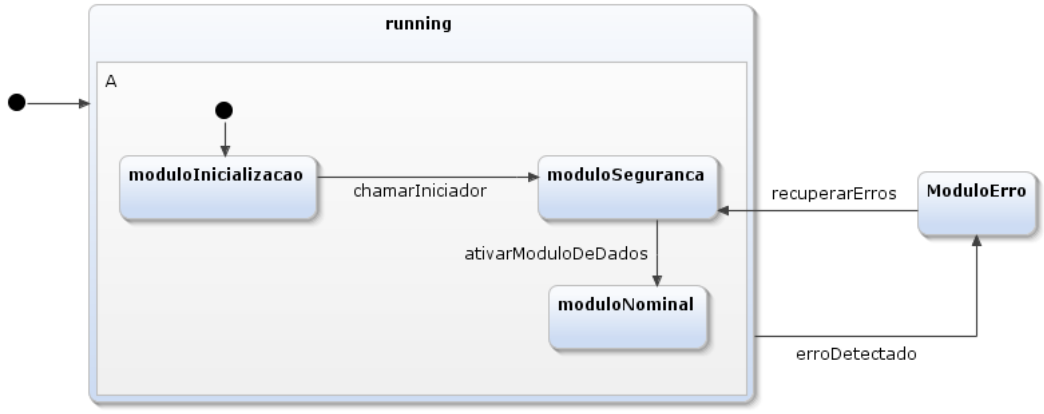


Figura 5.9: *Especificação do gerenciador de estados do software SWPDC.*

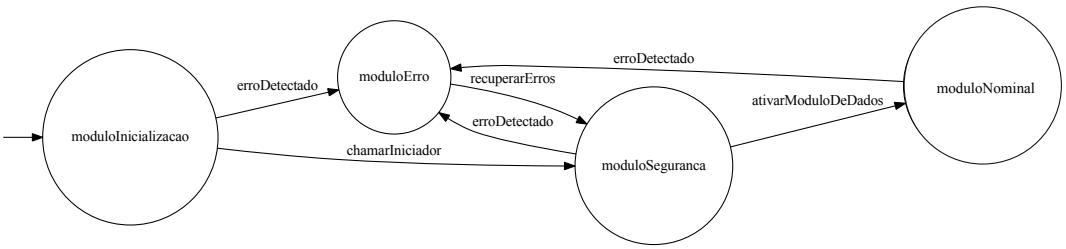


Figura 5.10: *Especificação do gerenciador de estados do software SWPDC no formato de uma máquina de estados finitos.*

```
6  //import java.io.IOException;
7  import suporte.Iniciador;
8  import fsm.FSM;
9
10 /**
11  *
12  * @author ahhul
13  */
14 public class SWPDC {
15
16     private GerenciadorDados gerenciadorDados;
17     private static int estadoPDC;
18
19     Iniciador iniciador;
20     Housekeeper hk;
21     static Temperatura temperatura;
22
23     public SWPDC(Housekeeper hk, GerenciadorDados gd) {
24         this.hk = hk;
25         gerenciadorDados = gd;
26     }
27
28     public static void main(String[] args) {
29         FSM fsm = new FSM();
30
31         Iniciador iniciador = new Iniciador();
32         iniciador.iniciar();
33
34         // Iniciador disparado, altera o evento e entra no estado de Iniciaçao
35         fsm.inicializar();
36
```

```
37          if (iniciador.m_relPOST == 1) {
38              estadoPDC = 1; /* fsm.informaEstado(); */
39          }
40
41          if (estadoPDC == 1) {
42              iniciador.ativarModuloDados();
43              // Estar no modulo de segurança e ativar Modulo de dados altera o
44              // evento
45              // E faz com que a maquina entre no estado Nominal
46              fsm.moduloDeDados();
47          }
48
49          temperatura = new Temperatura();
50          temperatura.executar(10);
51
52      }
53
54      /**
55       * Iniciação das tarefas de regime permanente (classes ativas).
56       */
57      public void iniciarTarefas() {
58          gerenciadorDados = GerenciadorDados.instanciar();
59          gerenciadorDados.iniciar();
60      }
61
62      public void finalizarTarefas() {
63          iniciador.finalizar();
64      }
65
66  }
67
68  }
```

Listagem 5.6: *Classe SWPDC.java do software SWPDC.*

```
1       package fsm;
2       import configuracao.ModoOperacaoEnum;
3
4       /**
5        *
6        * @author ahhul
7        */
8       public class FSM {
9
10          private ModoOperacaoEnum m_estadoAtual; // estado corrente da máquina
11          ModoOperacaoEnum moIniciacao, moNominal, moSeguranca, vazio; //0, 1,
                2.
12          private float m_numTransicoes; // quantidade de transicoes na tabela
                de transicoes
13          private int m_staFsm; // estado da maquina de estados. 1 não está em
                erro, 0 está em erro.
14
15
16          private EventoFsm evento = new EventoFsm();
17          private EstadoFsm estado = new EstadoFsm();
18
19          public FSM(){
20              evento.alteraEventoFsm(-1);
21              estado.alteraEstadoFsm(0);
22          }
```

```
23
24          /* Tabela de transição de Estados */
25          public ModoOperacaoEnum pesquisarTransicao(ModoOperacaoEnum atual){
26
27              ModoOperacaoEnum proximo = vazio;
28
29              /* Busca na tabela*/
30              if(atual == moIniciacao){
31                  proximo = moSeguranca;
32              }
33              if(atual == moSeguranca){
34                  proximo = moNominal;
35              }
36
37              return proximo;
38
39          }
40
41          public void atualizaEvento(int ocorrido){
42              evento.alteraEventoFsm(ocorrido);
43          }
44
45          public void inicializar(){
46              evento.alteraEventoFsm(0);
47              if((estado.retornaEstadoFsm() == 0)){
48                  /* Transicao modo de iniciaçao pro modo de segurança */
49                  if(pesquisarTransicao(moIniciacao) == moSeguranca){
50                      this.chamarIniciador();
51                  }
52              }else{
53                  /* Estado de erro da FSM */
54                  this.erroDetectado();
55              }
56              this.relatoEstado();
57          }
58
59          public void erroDetectado(){
60              estado.alteraEstadoFsm(3);
61          }
62
63          public void chamarIniciador(){
64              estado.alteraEstadoFsm(1);
65          }
66
67          public void ativarModuloDeDados(){
68              estado.alteraEstadoFsm(2);
69          }
70
71          public void moduloDeDados(){
72              evento.alteraEventoFsm(1);
73              if((estado.retornaEstadoFsm() == 1) ){
74                  /* Transicao modo de segurança pra modo nominal */
75                   if(pesquisarTransicao(moSeguranca) == moNominal){
76                     this.ativarModuloDeDados();
77                  }
78              }
79              else{
80                  /* Estado de erro da FSM */
81                  this.erroDetectado();
```

```
82              }
83              this.relatoEstado();
84          }
85
86
87          /* Retorna em int para verificações */
88          public int informaEstado(){
89              if(estado.retornaEstadoFsm() == 0)
90                  return 0;
91              if(estado.retornaEstadoFsm() == 1)
92                  return 1;
93              if(estado.retornaEstadoFsm() == 2)
94                  return 2;
95              if(estado.retornaEstadoFsm() == 3)
96                  return 3;
97              return -1;
98          }
99
100         /* Relato para execução do SWPDC */
101         public void relatoEstado(){
102
103             if(estado.retornaEstadoFsm() == 0){
104                 System.out.println("SWPDC em Modo de INICIAÇÃO");
105             }
106             if(estado.retornaEstadoFsm() == 1){
107                  System.out.println("SWPDC em Modo de SEGURANÇA");
108             }
109             if(estado.retornaEstadoFsm() == 2){
110                  System.out.println("SWPDC em Modo NOMINAL");
111             }
112             if(estado.retornaEstadoFsm() == 3){
113                  System.out.println("SWPDC em Estado de ERRO");
114             }
115         }
116
117
118     }
```

Listagem 5.7: *Classe FSM do SWPDC.*

Com a classe `ObserverSWPDC.java`, Listagem 5.8, chamamos o algoritmo `Flex`, e a partir dele obtemos o modelo que representa todo o possível comportamento do objeto `fsm` nesse software, Figura 5.11. As informações dessa mineração podem ser vistos na Tabela 5.8.

Software	Estados	Transições	Alfabeto	Tempo de Geração(ms)
SWPDC	10	14	3	369

Tabela 5.8: *Dados da mineração da máquina de estados finitos do sistema SWPDC.*

```
1  import br.usp.classes.Klass;
2
3  public class ObserverSWPDC{
4      public void examplo1(){
5          this.classFullName = "swpdc.SWPDC";
6          return Flex.generate(new Klass(classFullName, "main", "String[]"),"FSM
               ");
7      }
```

```
8 }
```

Listagem 5.8: *Classe ObserverSWPDC*

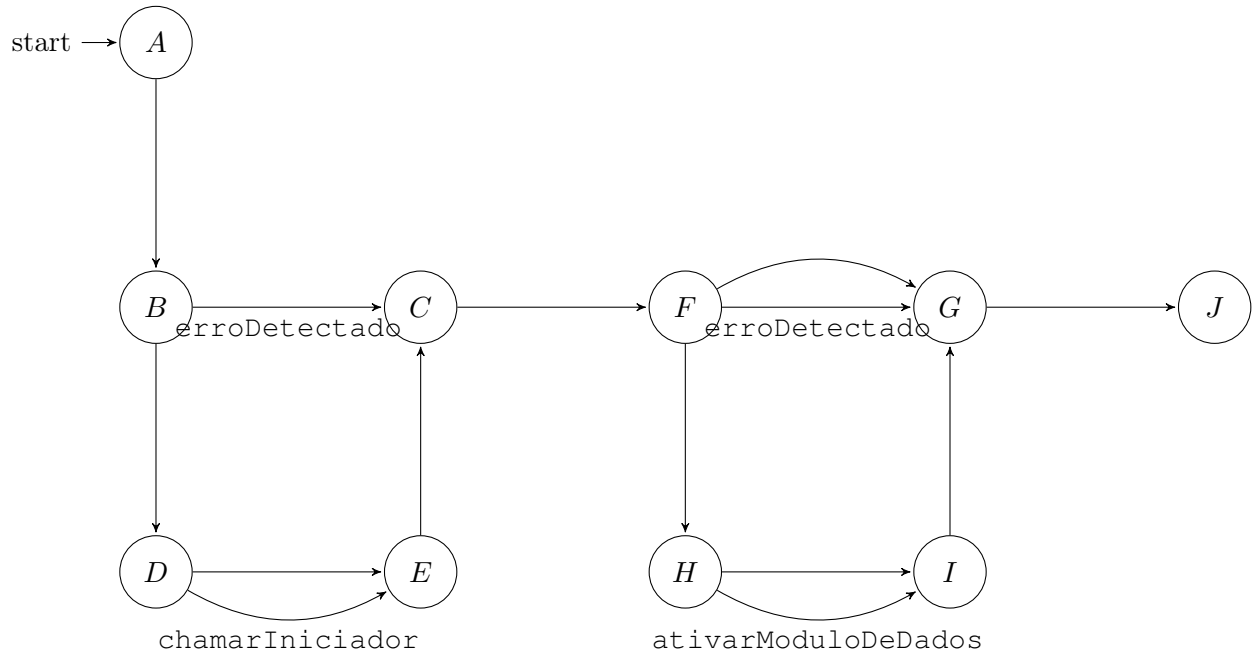


Figura 5.11: *Modelo minerado da implementação do SWPDC.*

5.5.2 Análise de Similaridade

Com os dois modelos prontos, realizamos então a comparação, Tabela 5.9, a qual nos apontou três PPFs, Tabela 5.10. A partir desses PPFs, analisamos o código e a especificação para entender o porquê dos PPFs. Identificamos então que embora os PPFs sejam apontados em nossa abordagem, no momento da execução do software, esses *traces* nunca irão acontecer pois há um `if` que garante que o sistema esteja no estado de inicialização para poder executar qualquer outra ação. Esses PPFs foram apontados pois, como dito anteriormente, nossa abordagem faz uma análise estática do código-fonte, e por isso não sabemos resultado das expressões lógicas, e portanto não tinha como o algoritmo Flex saber que a condição do `if` garantia que ele tinha que se encontrar no estado de inicialização para realizar as chamadas de métodos.

Esse estudo de caso nos mostrou que ainda se faz necessário enriquecer os modelos minerados, com informações do software em tempo de execução, através de análise dinâmica, buscando melhorar a precisão da nossa abordagem.

	Software	**Cenários**	**Alfabeto**	**Tempo de Simulação(ms)**	**PPF**
1	SWPDC	1	5	19	3

Tabela 5.9: *Resultado da Simulação SWPDC.*

5.6 Estudo de Caso com o Software SWPDC - 2

Nesse Estudo de caso, iremos trabalhar novamente com o software O SWPDC (Software Piloto Embarcado do Payload Data Handling Computer) [Ach13], porém, utilizando uma outra especifi-

	PPF
1	ativarModuloDeDados
2	erroDetectado - ativarModuloDeDados
3	erroDetectado - erroDetectado

Tabela 5.10: *PPF encontrados no SWPDC.*

cação que nós desenvolvemos para outra funcionalidade do software.

5.6.1 Definição dos Modelos

Neste estudo nós criamos um *statechart* olhando para implementação, para definir o comportamento ao que diz respeito a obter as amostras de temperatura pelo sistema. O *statechart* que fizemos pode ser visto na Figura 5.12 e o modelo da máquina de estados finitos obtida a partir do *statechart* pode ser visto na Figura 5.13.

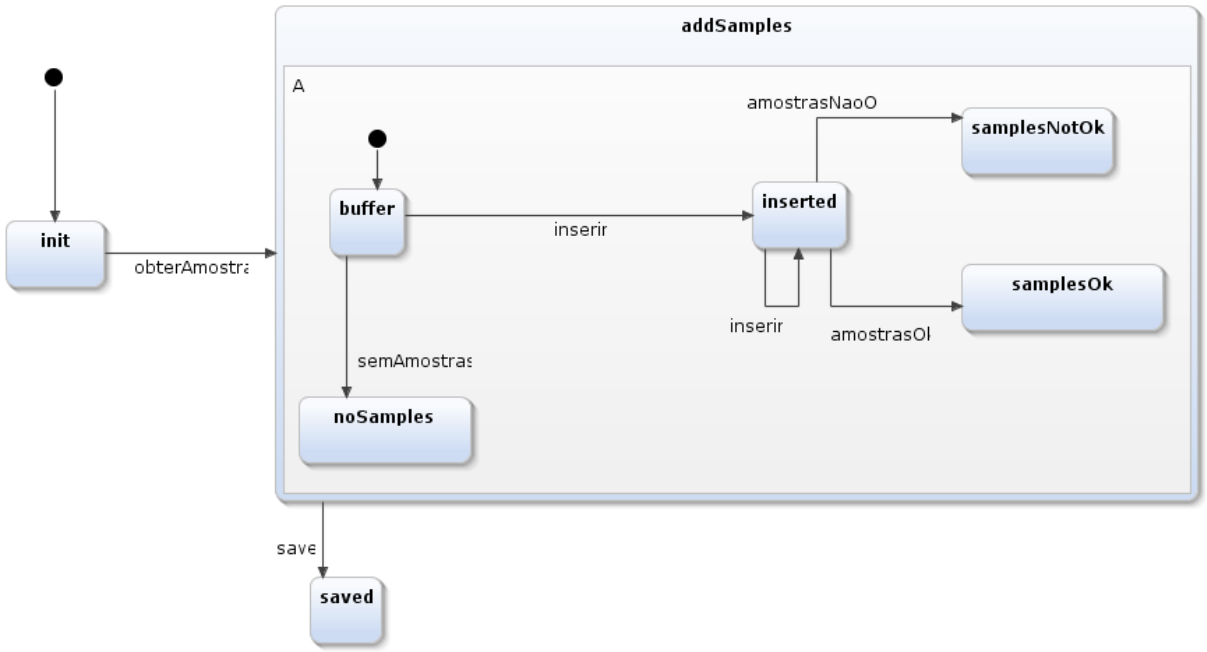


Figura 5.12: *Especificação do gerenciador de amostras de temperatura.*

Para a geração do modelo da implementação, observamos dentro da classe `obterAmostras.java` as modificações no próprio objeto, ou seja o `this`. A seguir podemos ver o código-fonte da classe `obterAmostras.java`, Listagem 5.9.

```
1     class ObterAmostras extends TimerTask{
2
3         @Override
4         public void run (){
5             m_bufTta.clear();
6             obterERelatarTemperatura();
7         }
8
9         public boolean temperaturaPertenceAoIntevalo(float temperatura) {
10            if(temperatura >= 10.0 && temperatura <= 40.0) {
```

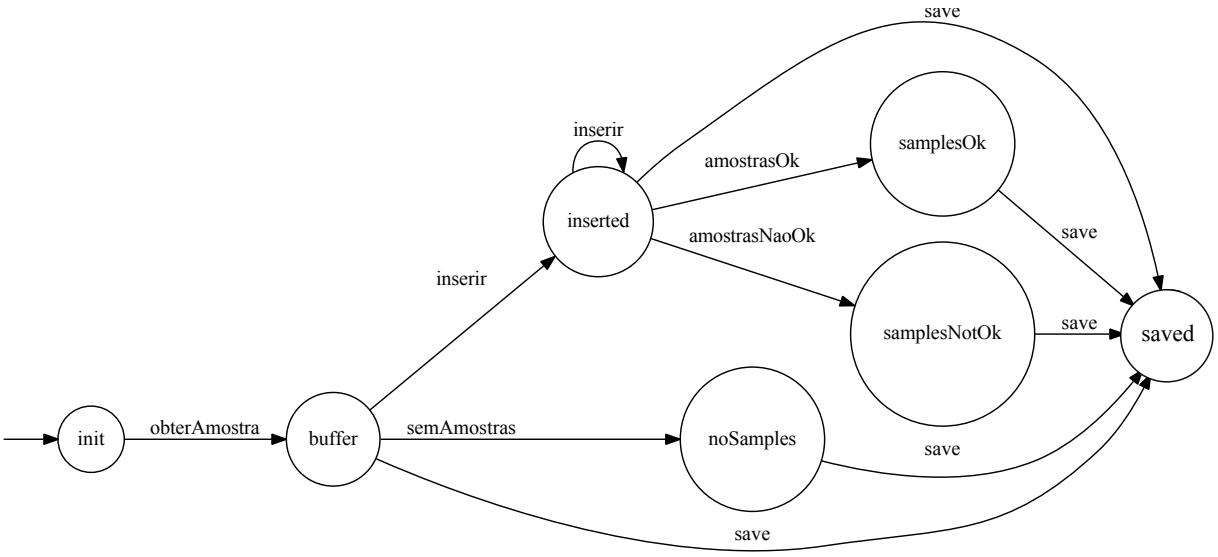


Figura 5.13: *Especificação do gerenciador de amostras de temperatura no formato de uma máquina de estados finitos.*

```
11                  return true;
12              }
13              else{
14                  System.out.println("temperatura " + temperatura + " fora do
                        intervalo.");
15                  return false;
16                }
17          }
18
19          public void obterAmostra() {
20              m_buffer = verificador.obterAmostras(amostraAtual);
21          }
22
23          private void inserir(float dado, int i) {
24              m_bufTta.inserir(dado, (byte)4, i);
25          }
26
27          private void obterERelatarTemperatura(){
28
29              comunicador.emitirComando("OBTER TEMPERATURA");
30
31              try {
32                  Thread.sleep((long)5000); //5 segundo
33              } catch (InterruptedException ex) {
34                  Logger.getLogger(Temperatura.class.getName()).log(Level.SEVERE
                        , null, ex);
35              }
36
37              System.out.println("Novas amostras de temperatura");
38              comunicador.guardarNoHistorico("Obtenção das novas amostras de
                    temperatura");
39
40              this.obterAmostra();
41
42              int todasAmostrasCorretas = 0;
43              if (m_buffer != null) {
44                  for (int i = 0; i < 12; i++){
45                       System.out.println(m_buffer[i]);
```

```
46                      if (temperaturaPertenceAoIntevalo (m_buffer[i]) ) {
47                          System.out.println("Pertenço ao intervalo");
48                          this.inserir(m_buffer[i], i);
49                          //System.out.println(m_bufTta.recuperar(i));
50
51                      } else{
52                          System.out.println("Não Pertenço ao intervalo");
53                          this.inserir(-1, i);
54                          System.out.println(m_bufTta.recuperar(i));
55
56                          todasAmostrasCorretas++;
57                      }
58                  }
59                  if (todasAmostrasCorretas == 0){
60                      this.amostrasOk();
61                  } else {
62                      this.amostrasNaoOk(todasAmostrasCorretas);
63                  }
64              }else{
65                  semAmostras();
66              }
67
68
69              try {
70                  this.save();
71              } catch (IOException ex) {
72                  Logger.getLogger(Temperatura.class.getName()).log(Level.SEVERE
                        , null, ex);
73              }
74              amostraAtual++;
75          }
76
77          /**
78           * @throws IOException
79           */
80          private void save() throws IOException {
81              hk.executar(m_bufTta);
82          }
83
84          /**
85           *
86           */
87          private void semAmostras() {
88              comunicador.guardarNoHistorico("Aguardando amostra...");
89              System.out.println("oiiiiiiii");
90          }
91
92          /**
93           * @param todasAmostrasCorretas
94           */
95          private void amostrasNaoOk(int todasAmostrasCorretas) {
96              System.out.println("Temperatura fora do intervalo detectada na
                    amostra " + amostraAtual + ".");
97              comunicador.guardarNoHistorico("Foram detectados " +
                    todasAmostrasCorretas + " dados errados durante o
                    armazenamento da amostra " + amostraAtual );
98          }
99
100         /**
```

```
101             *
102             */
103            private void amostrasOk() {
104                comunicador.guardarNoHistorico("Correto armazenamento da amostra "
                       + amostraAtual + " no buffer.");
105            }
106    }
```

Listagem 5.9: *Classe ObterAmostras*

Novamente com a classe `ObserverSWPDC.java`, Listagem 5.8, chamamos o algoritmo *Flex*, e a partir dele obtemos então o modelo que representa todo o possível comportamento do objeto no método `obterERelatarTemperatura()`, responsável por essa funcionalidade do software, Figura 5.11. As informações sobre a máquina de estados finitos obtida podem ser vistas na Tabela 5.11.

Software	Estados	Transições	Alfabeto	Tempo de Geração(ms)
SWPDC	9	13	5	524

Tabela 5.11: *Dados da mineração da máquina de estados finitos do sistema SWPDC - 2.*

```
1  import br.usp.classes.Klass;
2
3  public class ObserverSWPDC{
4      public void examplo2(){
5          this.classFullName = "dados.Temperatura.ObterAmostras";
6          return Flex.generate(new Klass(classFullName, "
               obterERelatarTemperatura"),"this");
7      }
8  }
```

Listagem 5.10: *Classe ObserverSWPDC*

5.6.2 Análise de Similaridade

Com os dois modelos prontos, realizamos então a comparação, Tabela 5.9, a qual nos apontou que não haviam PPFs nessa funcionalidade do software. Esse resultado já era esperado, visto que criamos a especificação, olhando para a implementação já feita. Contudo vale ressaltar que como agora possuímos a especificação, pode-se utilizar essa abordagem todas as vezes que houver modificações na implementação dessa funcionalidade, para dessa forma saber se o sistema ainda está obedecendo a especificação.

	Software	Cenários	Alfabeto	Tempo de Simulação(ms)	PPF
1	SWPDC	1	6	40	0

Tabela 5.12: *Resultado da Simulação SWPDC.*

5.7 Conclusão

Os estudos de caso realizados nos permitiram checar se os sistemas estavam de acordo com suas especificações. O resultado da simulação nos apontou PPFs para que pudéssemos checar manualmente os pontos onde o sistema poderia estar fazendo algo fora da especificação, e quando não

encontrou nenhum PPF, nos garantiu que todos os *traces* realizados estavam previstos na especificação.

No próximo capítulo apresentamos com mais detalhes as conclusões tiradas a partir da análise dos resultados obtidos, assim como as limitações que foram encontradas em nossa abordagem durante sua aplicação.

Capítulo 6

Conclusões

O uso de sistemas baseados em softwares aumentou, de forma vertiginosa, no cotidiano das pessoas sendo, em muitos casos, responsáveis por tarefas de extrema criticidade. Por isso, é vital que estes softwares operem dentro dos parâmetros especificados pelos seus projetos a fim de não apresentarem falhas que possam vir a causar perdas de bens e/ou vidas. Logo, devido à importância da identificação de falhas nos sistemas, a indústria de software utiliza duas metodologias distintas, mas complementares, para assegurar o máximo de qualidade a seus produtos. Estas metodologias são teste de software e verificação formal. Como colocado anteriormente neste trabalho, ambas as técnicas são custosas e dependentes da "interpretação" detalhada da especificação, feita pelo desenvolvedor, para a construção das malhas de teste (teste de software) ou modelos formais (verificação formal), o que torna o processo uma atividade manual e dependente da experiência do desenvolvedor.

Assim, sensibilizados pelas questões acima levantadas, apresentamos uma metodologia que compara diretamente a especificação de um software com sua codificação. Nossa metodologia compara um modelo *statechart*, com a especificação inicial do software, com uma máquina de estados finitos obtida por meio de uma análise estática do código-fonte. Se forem constatadas diferenças entre os dois modelos, admitimos que há fortes indícios de que um erro foi encontrado e que este erro pode originar falhas. Por sua vez, a diferença entre os modelos se apresenta na forma de caminhos, *traces* de execução, presentes em um e ausentes no outro. Por fim, para determinar se os *traces* divergentes representam ou não erros no software, apresentamos essas diferenças ao desenvolvedor, o qual pode caracterizar a discrepância encontrada como um erro que deve ser corrigido, uma incompletude da especificação ou um fluxo sem relevância para o funcionamento da aplicação.

Desta forma, os objetivos atingidos por este trabalho foram:

1. Colaboração com os estudos acadêmico-científicos que visam ampliar a compreensão sobre o fenômeno da falha de software e os meios para eliminar e/ou mitigar seus efeitos na produção de software;

2. Apresentação de uma metodologia técnico-científica que permite a identificação de erros de software ainda em fases iniciais do desenvolvimento, a qual possui potencial para reduzir custos de desenvolvimento, aumentar a automação do processo de verificação de software e expandir a efetividade e eficácia da capacidade de localização de falhas;

3. Construção de um protótipo de ferramenta, o qual implementa os elementos técnicos-científicos de nossa metodologia permitindo auferir os reais resultados obtidos e fornece perspectivas para novas melhorias, ver Seção 6.1;

4. Produção de resultados experimentais relevantes, os quais serão organizados para serem publicados em um periódico técnico-científico, com o intuito de socializar com meio acadêmicos os conhecimentos produzidos.

6.1 Discussão dos Resultados Obtidos

Dentre os trabalhos que tratam de mineração de máquinas de estados finitos encontrados no período da revisão bibliográfica, nenhum dos algoritmos fazia a mineração da maneira que queríamos, retornando uma máquina de estados finitos onde cada transição representa uma mudança que pode ocorrer com um objeto especificado. Por essa razão foi necessário definir e implementar nosso próprio algoritmo de mineração, algoritmo Flex, o qual se mostrou com tempo de execução bastante eficiente, como pode ser visto nos estudos de caso, embora ainda possua limitações como não reconhecer o bloco *try/catch* do Java, além de precisar ser testado de forma exaustiva. Também foi necessário definir e implementar nossa própria relação de simulação, de forma a identificar de maneira automática a diferença entre as máquinas de estados finitos. Essa relação foi baseada na *simulation order*, porém foi necessário fazer uma adaptação para ignorar as transições τ que haviam sido inseridas no modelo da implementação. Esses dois algoritmos foram os principais componentes para aplicação de nossa abordagem, a qual foi colocada em prática através dos estudos de caso realizados.

Os estudos de caso realizados neste trabalho utilizaram sistemas que possuem as seguintes características:

- implementados na linguagem Java;
- possuem especificação modelada em *statechart*, e
- o modelo deve apresentar informações que são próximas a implementação.

Após a realização dos estudos de caso e análise dos PPFs identificados, pudemos observar que:

1. O experimento permitiu identificar *traces* que não eram permitidos na especificação, o que vem ao encontro de nosso pressupostos teóricos. As ocorrências desses *traces*, neste experimento, já eram previstas e o tratamento para estes casos se fazia realmente necessário.
2. O experimento apresentou, também, o comportamento teórico esperado quando o sistema alvo não apresentava diferenças entre os modelos da especificação e o baseado no código-fonte.
3. Por fim, foi possível constatar que o modelo da aplicação construído na forma de máquinas de estados finitos facilitava o processo de identificar quais requisitos determinados pela especificação não tinham sido efetivamente implementados.

Durante a realização do experimento também foram encontradas algumas dificuldades para aplicação da metodologia proposta neste trabalho. As dificuldades encontradas foram:

1. Especificações em alto nível: muitos dos *statecharts* pesquisados não possuíam uma granularidade que os aproximasse do código-fonte, em outras palavras, eram modelos gerais do software e não modelos de orientação da codificação.
2. Códigos-fonte não modularizados: o código-fonte de alguns dos sistemas selecionados para o experimento tiveram de ser alterados a fim de que conjuntos de instruções que encontramos no código pudessem ser equivalente a uma evento do *statechart*.
3. Tempo de aplicação da abordagem: o tempo para a aplicação de nossa abordagem sofreu um acréscimo pela necessidade de utilizar a ferramenta GTSC para conversão do *statechart* em máquina de estados finitos, visto que a importação do *statechart* no GTSC deve ser feita manualmente. Entretanto, levando em consideração somente nossos algoritmos de mineração e de simulação das máquinas de estados finitos, obtivemos bons tempos de execução, ainda mais se comparado a outros trabalhos como o do Fink et al. [FYD+08] nos quais os algoritmos possuem latência de vários segundos para serem executados. Um fator que influencia diretamente para essa diferença de tempo é que o trabalho de Fink faz uma varredura total no

sistema buscando os locais onde minerar, enquanto nós já apontamos qual parte do software deve ser observada. Outro fator que impacta nesse tempo são as poucas modificações que fazemos no modelo que é minerado do software, visto que a única alteração feita é a omissão dos métodos excessivos.

4. Problema de Nomeação: outro ponto que dificultou a aplicação da nossa abordagem foi a diferença entre os nomes das ações no modelo da especificação e o nome das ações no sistema implementado. Esse problema já foi constatado no trabalho de Foster et al. [FUMK03], pois os desenvolvedores podem utilizar diferentes convenções na nomenclatura dos métodos da implementação. Esse problema foi contornado por nós através de um mapeamento, quando necessário, entre as ações do *statechart* e os métodos da implementação.

5. Ocorrência de falsos positivos: A partir dos estudos de casos foram apontados alguns PPFs, contudo, dentre os PPFs encontrados tivemos um grande número de falsos positivos, ou seja, esses PPFs não acarretariam de fato em falhas no sistema. Esse tipo limitação já era esperada, visto que outros trabalhos como o de Fink et al. [FYD+08], onde são testados vários algoritmos, também mostram uma alta taxa de falsos positivos. Em nosso trabalho isso se deu, em grande parte, por minerarmos os modelos da implementação somente de maneira estática, enquanto fatores como operações lógicas em estruturas de decisão e/ou repetição, que influenciam diretamente no fluxo do sistema, só podem ser entendidas de maneira dinâmica. Isso ficou bem claro no estudo de caso do gerenciador dos modos de operação do SWPDC, Seção 5.5, no qual os três PPFs encontrados nunca ocorreriam realmente na execução do software, pois a verificação já estava sendo feita com um `if`.

Com base nas observações feitas em relação ao experimento e com base nas dificuldades encontradas, é possível concluir que a metodologia aqui apresentada possui grande potencial para tornar mais eficiente e eficaz a detecção de erros que podem gerar falhas. Como constatado, nossa implementação efetivamente identificava variações nos dois modelos do software que podem ser entendidas como caminhos indevidamente implementados. Além disso, as dificuldades encontradas levam à conclusão de que existe efetivamente um lapso entre a modelagem de uma aplicação e a efetiva codificação. Este lapso pode ser expresso pela falta de modelos que efetivamente orientem e regrem a construção do software. Decisões como agrupamento de código e nomeação de métodos ficam a cargo do desenvolvedor e, por consequência, são fortemente afetadas pela experiência deste. Tal situação resulta em uma documentação, modelagem, que não expressa a realidade do software que efetivamente foi construído o que sem dúvida trará consequências futuras negativas em termos de manutenção da aplicação e detecção de falhas.

6.2 Trabalhos Futuros

Existe um conjunto de trabalhos que poderiam ser realizados buscando contornar as limitações citadas anteriormente, além de melhorar a precisão dos resultados e aprimorar a usabilidade da abordagem, são eles:

- Utilizar análise dinâmica, de forma a enriquecer os modelos minerados, identificando *traces* mais recorrentes e/ou *traces* que nunca ocorrem, assim como permitir a observação dos blocos de tratamento de exceções (*try/catch*), o que não foi viável só com análise estática;

- Gerar automaticamente o código-fonte de uma *interface* a partir da modelagem *statechart*, para que dessa forma os desenvolvedores possam implementar tais interfaces, tirando a necessidade de fazer um mapeamento dos nomes das ações do *statechart* com os métodos do código-fonte;

- Implementar o algoritmo de transformação do *statechart* para máquina de estados finitos, Vijaykumar et al [VCA02], tirando assim a necessidade dos outros dois conversores (SCT2PCML

e XML2AUTOMATO), além de unificar todo o processo em uma só ferramenta, que no caso seria a Yakindu.

Apêndice A

Frameworks e Softwares Utilizados

A.1 Eclipse

Eclipse IDE [ecl] é uma das IDEs mais populares quando se fala de desenvolvimento de código Java. Sua versatilidade, funcionalidades e facilidade de uso a tornam uma ferramenta ideal para desenvolvimento de sistemas Java. Utilizamos o Eclipse como IDE para o desenvolvimento dos códigos a serem analisados, entretanto, outras IDE's como NetBeans [net] e IntelliJ IDEA [int] podem ser usadas, visto que a IDE usada para desenvolver o código não interfere em nossa abordagem.

A.2 Recoder

Para realizar a análise estática do sistema, optamos por utilizar o *framework* Recoder [rec], sendo ele um *framework* de código aberto, escrito na linguagem Java, que nos permite fazer análise em cima de códigos-fonte. Nós utilizamos este *framework* na construção da máquina de estados finitos que representa a implementação do software, pois dado o código-fonte de uma aplicação (.java) que queremos analisar, o Recoder faz uma análise léxica e sintática, e nos retorna as estruturas e comandos contidos nele. Com a informação das estruturas de controle, combinada com as definições para geração do grafo de fluxo de controle feitas na Seção 3.2.1, nos foi permitido construir a máquina de estados finitos do comportamento do sistema.

A.3 Yakindu

Desenvolvida pela empresa itemis, Yakindu [yak] é uma ferramenta de código aberto, baseada na IDE Eclipse, e focada no desenvolvimento de sistemas embarcados. Ele é composto de duas partes, o SCT e o Damos. Nós utilizamos a parte do SCT, visto que esse possui um editar gráfico, simples, porém bem elaborado que permite desenhar os *statecharts* para nossa especificação. Todos os *statecharts* apresentados neste trabalho foram desenhados utilizando o editor gráfico dessa ferramenta.

A.4 GTSC

O software GTSC (Geração Automática de Casos de Teste Baseada em *Statecharts*) [SVG+08] é uma ferramenta que permite modelar o comportamento de softwares em forma de *statecharts*, e automatizar geração de casos de testes. Para conseguir gerar esses casos de testes, o GTSC dispõe de uma funcionalidade que converte o *statechart* em máquina de estados finitos plana, funcionalidade essa que implementa as definições dada por Vijay at al [VCAA06] e explicadas neste trabalho na Seção 2.3.2. Nós fazemos uso do GTSC de modo a converter o nosso modelo *statechart* feito no

YAKINDU SCT em uma máquina de estados finitos.

Apêndice B

Transformações nos Modelos

B.1 SCT para PCML

Especificação da classe `java.net.Socket` no formato SCT.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3
    .org/2001/XMLSchema-instance" xmlns:notation="http://www.eclipse.org/gmf/runtime
    /1.0.2/notation" xmlns:sgraph="http://www.yakindu.org/sct/sgraph/2.0.0">
  <sgraph:Statechart xmi:id="_AvA7QJ2eEeW5bPZ0kvOKTw" specification="interface Socket
      :&#xA;&#x9;in event connect&#xA;&#x9;in event getInputStream&#xA;&#x9;in event
      getOutputStream&#xA;&#x9;in event close" name="Socket">
    <regions xmi:id="_AvCJYp2eEeW5bPZ0kvOKTw" name=",">
      <vertices xsi:type="sgraph:Entry" xmi:id="_AvHo8Z2eEeW5bPZ0kvOKTw">
        <outgoingTransitions xmi:id="_ZXUmYJ2hEeW5bPZ0kvOKTw" specification="" target=
            "_G_bJYJ2eEeW5bPZ0kvOKTw"/>
      </vertices>
      <vertices xsi:type="sgraph:State" xmi:id="_G_bJYJ2eEeW5bPZ0kvOKTw" name="initial
          " incomingTransitions="_ZXUmYJ2hEeW5bPZ0kvOKTw _ct5VCMzSEeWUWvprLaCWbA">
        <outgoingTransitions xmi:id="_rH4lUJ2eEeW5bPZ0kvOKTw" specification="close"
            target="_Sr6eUJ2hEeW5bPZ0kvOKTw"/>
        <regions xmi:id="_bE5C0J2eEeW5bPZ0kvOKTw" name="A">
          <vertices xsi:type="sgraph:State" xmi:id="_fgx_UJ2eEeW5bPZ0kvOKTw" name="
              connected" incomingTransitions="_vIsIUJ2eEeW5bPZ0kvOKTw
              _bsjYYJ2hEeW5bPZ0kvOKTw">
            <outgoingTransitions xmi:id="_vIsIUJ2eEeW5bPZ0kvOKTw" specification="
                getInputStream, &#xA;getOutputStream" target="_fgx_UJ2eEeW5bPZ0kvOKTw"
                />
          </vertices>
          <vertices xsi:type="sgraph:Entry" xmi:id="_looHUJ2eEeW5bPZ0kvOKTw">
            <outgoingTransitions xmi:id="_36mC0J2eEeW5bPZ0kvOKTw" specification=""
                target="_Wl6OsJ2hEeW5bPZ0kvOKTw"/>
          </vertices>
          <vertices xsi:type="sgraph:State" xmi:id="_Wl6OsJ2hEeW5bPZ0kvOKTw" name="
              unconnected" incomingTransitions="_36mC0J2eEeW5bPZ0kvOKTw">
            <outgoingTransitions xmi:id="_bsjYYJ2hEeW5bPZ0kvOKTw" specification="
                connect" target="_fgx_UJ2eEeW5bPZ0kvOKTw"/>
          </vertices>
        </regions>
      </vertices>
      <vertices xsi:type="sgraph:State" xmi:id="_Sr6eUJ2hEeW5bPZ0kvOKTw" name="closed"
          incomingTransitions="_rH4lUJ2eEeW5bPZ0kvOKTw">
        <outgoingTransitions xmi:id="_ct5VCMzSEeWUWvprLaCWbA" specification="oncycle"
            target="_G_bJYJ2eEeW5bPZ0kvOKTw"/>
      </vertices>
    </regions>
  </sgraph:Statechart>
</xmi:XMI>
```

Especificação da classe `java.net.Socket` convertida de SCT para PCML.

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3  <Pcml Title="untitledModel" Date="2015-12-12">
4    <Info>
5      <Author>
6        <Name>Kelvin</Name>
7      </Author>
8    </Info>
9    <States>
10     <Root Name="top" Type="XOR" Default="initial">
11       <State Name="closed" Type="BASIC" />
12       <State Name="initial" Type="XOR" Default="unconnected">
13         <State Name="connected" Type="BASIC" />
14         <State Name="unconnected" Type="BASIC" />
15       </State>
16     </Root>
17   </States>
18   <Events>
19     <Stochastic Name="getOutputStream" Value="1.0" />
20     <Stochastic Name="close" Value="1.0" />
21     <Stochastic Name="getInputStream" Value="1.0" />
22     <Stochastic Name="connect" Value="1.0" />
23   </Events>
24   <Transitions>
25     <Transition Source="initial" Destination="closed" Event="close" />
26     <Transition Source="unconnected" Destination="connected" Event="connect" />
27     <Transition Source="connected" Destination="connected" Event="getInputStream" />
28     <Transition Source="connected" Destination="connected" Event="getOutputStream" />
29   </Transitions>
30 </Pcml>
```

B.2 PCML para Máquina de estados finitos

A máquina de estados finitos construída pelo software GTSC a partir do *statechart* no formato PCML é gerada em uma arquivo XML. A seguir temos o XML obtido a partir da conversão do *statechart* da especificação da classe `java.net.Socket`.

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3  <?xml-stylesheet type="text/xsl" href="mfeeTesteX.xsl"?>
4  <MFEE>
5      <STATES>
6          <STATE NAME="unconnected" TYPE="inicial"/>
7          <STATE NAME="closed" TYPE="normal"/>
8          <STATE NAME="connected" TYPE="normal"/>
9          <STATE NAME="unconnected" TYPE="final"/>
10     </STATES>
11     <EVENTS>
12         <EVENT VALUE="1" NAME="close"/>
13         <EVENT VALUE="1" NAME="connect"/>
14         <EVENT VALUE="1" NAME="oncycle"/>
15         <EVENT VALUE="1" NAME="getInputStream"/>
16         <EVENT VALUE="1" NAME="getOutputStream"/>
17     </EVENTS>
18     <INPUTS>
19         <INPUT EVENT="close"/>
20         <INPUT EVENT="connect"/>
21         <INPUT EVENT="oncycle"/>
22         <INPUT EVENT="getInputStream"/>
23         <INPUT EVENT="getOutputStream"/>
24     </INPUTS>
```

```
25      <OUTPUTS>
26      </OUTPUTS>
27      <TRANSITIONS>
28          <TRANSITION SOURCE="unconnected" DESTINATION="closed">
29              <INPUT INTERFACE="L">close</INPUT>
30                  <OUTPUT></OUTPUT>
31      </TRANSITION>
32          <TRANSITION SOURCE="unconnected" DESTINATION="connected">
33              <INPUT INTERFACE="L">connect</INPUT>
34                  <OUTPUT></OUTPUT>
35      </TRANSITION>
36          <TRANSITION SOURCE="closed" DESTINATION="unconnected">
37              <INPUT INTERFACE="L">oncycle</INPUT>
38                  <OUTPUT></OUTPUT>
39      </TRANSITION>
40          <TRANSITION SOURCE="connected" DESTINATION="closed">
41              <INPUT INTERFACE="L">close</INPUT>
42                  <OUTPUT></OUTPUT>
43      </TRANSITION>
44          <TRANSITION SOURCE="connected" DESTINATION="connected">
45              <INPUT INTERFACE="L">getInputStream</INPUT>
46                  <OUTPUT></OUTPUT>
47      </TRANSITION>
48          <TRANSITION SOURCE="connected" DESTINATION="connected">
49              <INPUT INTERFACE="L">getOutputStream</INPUT>
50                  <OUTPUT></OUTPUT>
51      </TRANSITION>
52      </TRANSITIONS>
53  </MFEE>
```

Referências Bibliográficas

[Ach13] Camila Achutti. Mission critical software validation and verification, 2013. 52, 57

[AO08] Paul Ammann e Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008. 1, 23, 24, 25, 27

[BK+08] Christel Baier, Joost-Pieter Katoen et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008. xi, 1, 34, 35, 36

[BRJ97] Grady Booch, James Rumbaugh e Ivar Jacobson. Uml notation guide, version 1.1. *Rational Software Corporation, Santa Clara, CA*, 2, 1997. 11

[CBGU13] Guido De Caso, Victor Braberman, Diego Garbervetsky e Sebastian Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):25, 2013. 3, 6, 7

[CBM89] Olivier Coudert, Christian Berthet e Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. Em *International Conference on Computer Aided Verification*, páginas 365–373. Springer, 1989. 10

[CD05] Michelle L Crane e Juergen Dingel. Uml vs. classical vs. rhapsody statecharts: Not all models are created equal. Em *Model Driven Engineering Languages and Systems*, páginas 97–112. Springer, 2005. 11

[dCBGU11] Guido de Caso, Víctor Braberman, Diego Garbervetsky e Sebastián Uchitel. Program abstractions for behaviour validation. Em *Software Engineering (ICSE), 2011 33rd International Conference on*, páginas 381–390. IEEE, 2011. 6, 7

[DHWT91] David L Dill, Alan J Hu e Howard Wong-Toi. Checking for language inclusion using simulation preorders. Em *Computer Aided Verification*, páginas 255–265. Springer, 1991. 36

[DSD05] Doron Drusinsky, Man-Tak Shing e Kadir Demir. Test-time, run-time, and simulation-time temporal assertions in rsp. Em *Rapid System Prototyping, 2005.(RSP 2005). The 16th IEEE International Workshop on*, páginas 105–110. IEEE, 2005. 6

[ecl] Eclipse. http://www.eclipse.org/. Acessado: 02-03-2016. 42, 67

[FUMK03] Howard Foster, Sebastian Uchitel, Jeff Magee e Jeff Kramer. Model-based verification of web service compositions. Em *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, páginas 152–161. IEEE, 2003. 7, 65

[FYD+08] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam e Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008. 5, 6, 7, 43, 64, 65

[GL94] Orna Grumberg e David E Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994. 36

[GLLS07] Orna Grumberg, Martin Lange, Martin Leucker e Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Information and Computation*, 205(8):1130–1148, 2007. 33

[gor] Gordon college. http://www.math-cs.gordon.edu/courses/cs211/ATMExample/. Acessado: 12-03-2016. 47

[Han15] Simone Hanazumi. Geração de propriedades sobre programas java a partir de objetivos de teste, 2015. 50

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987. 2, 3, 7, 11

[HK04] David Harel e Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). Em *Integration of Software Specification Techniques for Applications in Engineering*, páginas 325–354. Springer, 2004. 11

[HP85] David Harel e Amir Pnueli. *On the development of reactive systems*. Springer, 1985. 11

[HPSS87] David Harel, Amir Pnueli, Jeanette Schmidt e Rivi Sherman. On the formal semantics of statecharts. 1987. 11, 12

[int] Intellij idea. https://www.jetbrains.com/idea/. Acessado: 02-03-2016. 67

[JS08] Pallavi Joshi e Koushik Sen. Predictive typestate checking of multithreaded java programs. Em *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, páginas 288–296. IEEE Computer Society, 2008. 5

[KBP+10] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia e Nenad Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. Em *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, páginas 179–182. IEEE, 2010. 4, 5

[KKN15] M Raveendra Kumar, Raghavan Komondoor e S Narendran. Static analysis of file-processing programs using file format specifications. *arXiv preprint arXiv:1501.04730*, 2015. 6, 7

[LGW09] Claire Le Goues e Westley Weimer. *Specification mining with few false positives*. Springer, 2009. 5

[LKHL11] David Lo, Siau-Cheng Khoo, Jiawei Han e Chao Liu. *Mining Software Specifications: Methodologies and Applications*. CRC Press, 2011. 2, 3, 21, 22

[LL13] Yongming Li e Lijun Li. Model checking of linear-time properties based on possibility measure. *Fuzzy Systems, IEEE Transactions on*, 21(5):842–854, 2013. 34

[MP92] Zohar Manna e Amir Pnueli. *The temporal logic of reactive and concurrent systems: specifications*, volume 1. Springer Science & Business Media, 1992. 11

[MSY12] Alon Mishne, Sharon Shoham e Eran Yahav. Typestate-based semantic code search over partial programs. Em *ACM SIGPLAN Notices*, volume 47, páginas 997–1016. ACM, 2012. 3

[net] Netbeans ide. https://netbeans.org/. Acessado: 02-03-2016. 67

[PSYY13] Hila Peleg, Sharon Shoham, Eran Yahav e Hongseok Yang. Symbolic automata for static specification mining. Em *Static Analysis*, páginas 63–83. Springer, 2013. 3, 5

[rec] Recoder. https://sourceforge.net/projects/recoder/. Acessado: 02-03-2016. 23, 42, 67

[SG03] Sharon Shoham e Orna Grumberg. A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. Em *Computer Aided Verification*, páginas 275–287. Springer, 2003. 33

[sta] Statecharts examples. http://statecharts.net/engl_bsps.htm. Acessado: 14-03-2016. 50

[SVG+08] Valdivino Santiago, Nandamudi Lankalapalli Vijaykumar, Danielle Guimarães, Ana Silvia Amaral e Érica Ferreira. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. Em *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, páginas 63–72. IEEE, 2008. 14, 67

[SY86] Robert E Strom e Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *Software Engineering, IEEE Transactions on*, (1):157–171, 1986. 6

[SYFP08] Sharon Shoham, Eran Yahav, Stephen J Fink e Marco Pistoia. Static specification mining using automata-based abstractions. *Software Engineering, IEEE Transactions on*, 34(5):651–666, 2008. 5

[TPBL95] QM Tan, A Petrenko, G v Bochmann e G Luo. *Testing trace equivalence for labeled transition systems*. Université de Montréal, Département d'informatique et de recherche opérationnelle, 1995. 34

[VCA02] Nandamudi Lankalapalli Vijaykumar, Solon Venancio de Carvalho e Vakulathil Abdurahiman. On proposing statecharts to specify performance models. *International Transactions in Operational Research*, 9(3):321–336, 2002. 16, 17, 65

[VCAA06] Nandamudi Lankalapalli Vijaykumar, SV Carvalho, VMB Andrade e Vakulathil Abdurahiman. Introducing probabilities in statecharts to specify reactive systems for performance analysis. *Computers & operations research*, 33(8):2369–2386, 2006. xi, 14, 16, 17, 19, 67

[Was10] Andrzej Wasylkowski. *Object usage: Patterns and anomalies*. Tese de Doutorado, PhD thesis, Saarland University, 2010. 5, 7

[WML02] John Whaley, Michael C Martin e Monica S Lam. Automatic extraction of object-oriented component interfaces. Em *ACM SIGSOFT Software Engineering Notes*, volume 27, páginas 218–228. ACM, 2002. 5

[WZL07] Andrzej Wasylkowski, Andreas Zeller e Christian Lindig. Detecting object usage anomalies. Em *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, páginas 35–44. ACM, 2007. 5, 7

[yak] Yakindu. https://www.itemis.com/en/yakindu/statechart-tools/. Acessado: 02-03-2016. 14, 41, 67