Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Matheus Takeshi Yamakawa Ikeda

Teste de software baseado em lógica temporal linear em sistemas reativos

João Monlevade
2018

Matheus Takeshi Yamakawa Ikeda

Teste de software baseado em lógica temporal linear em sistemas reativos

Monografia apresentada ao curso de Engenharia de Computação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II"

Orientador: Elton Maximo Cardoso.

João Monlevade

2018

I262t Ikeda, Matheus Takeshi Yamakawa.
Teste de software baseado em lógica temporal linear em sistemas reativos [manuscrito] / Matheus Takeshi Yamakawa Ikeda. - 2018.

44f.:

Orientador: Prof. MSc. Elton Maximo Cardoso.

Monografia (Graduação). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Departamento de Computação e Sistemas de Informação.

1. Engenharia de software. 2. Software - Testes. 3. Software - Validação. 4. Lógica de computador. 5. Sistemas embarcados (Computadores). I. Cardoso, Elton Maximo. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004.415.5

Catalogação: ficha.sisbin@ufop.edu.br

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS
COLEGIADO DE ENGENHARIA DA COMPUTAÇÃO

ATA DE DEFESA

Aos 19 dias do mês de julho de 2018, às 11 horas e, na Sala E204 do ICEA, foi realizada a defesa de Monografia pelo aluno **Matheus Takeshi Yamakawa Ikeda,** sendo a Comissão Examinadora constituída pelos professores: Prof. Msc. Elton Máximo Cardoso, Prof. Msc. Alexandre Magno de Souza e Prof. Msc. Bruno Cerqueira Hott. O candidato apresentou a monografia intitulada: *"Teste de software baseado em lógica temporal linear em sistemas reativos".* A comissão examinadora deliberou, por unanimidade, pela aprovação do candidato, com nota 10 (dez pontos), concedendo-lhe o prazo de 15 dias para incorporação das alterações sugeridas ao texto final.

João Monlevade, 19 de julho de 2018.

Elton M. Cardoso
Prof. Msc. Elton Máximo Cardoso
Professor Orientador/Presidente

Alexandre M. Sou
Prof. Msc. Alexandre Magno de Souza
Professor Convidado

Prof. Msc. Bruno Cerqueira Hott
Professor Convidado

Matheus Ikeda
Matheus Takeshi Yamakawa Ikeda
Discente

Este trabalho é dedicado a Luciane e Wilson

Agradecimentos

Agradeço aos meus pais pelo enorme apoio, paciência e consolo durante essa trajetória, que nunca mediram esforços para priorizar minha educação. Sei o quanto vocês se doaram para a realização deste sonho e é graças a vocês que concluo mais esta etapa da minha vida.

Agradeço a universidade pela oportunidade e ao meu orientador pela ajuda e disposição na elaboração e revisão deste trabalho.

Também sou grato aos meus amigos pelos incentivos e bons momentos que tornaram esta jornada mais leve e feliz.

"It is the quality of one's convictions that determines success, not the number of followers."

— J.K. Rowling,
in: Harry Potter and the Deathly Hallows.

Resumo

Testes são certamente a abordagem mais utilizada por desenvolvedores de todo o mundo para a garantia da qualidade de software. Porém, a construção e execução de testes é uma atividade dispendiosa, responsável por cerca de 50% do custo do desenvolvimento de um sistema. Atualmente, sistemas embarcados são ubíquos na sociedade moderna e estão presentes nos mais diversos dispositivos eletrônicos onde a corretude e segurança são fatores críticos, como aplicações bancárias e dispositivos médicos. Por sua execução ser determinada por eventos externos, testar sistemas embarcados é uma tarefa complexa, sendo muitas vezes feita com o sistema em produção, o que pode acarretar o aumento do custo do desenvolvimento do software. Além das dificuldades para construção de testes, sistemas reativos não são facilmente especificados utilizando fórmulas da lógica de primeira ordem, uma vez que esta não possui mecanismos para representar, de forma simples, a relação temporal entre ações do sistema. Tais formulações são feitas de maneira natural utilizando as chamadas lógicas temporais, em que a noção de tempo é explícita na semântica de fórmulas. O presente trabalho representa uma contribuição tecnológica por fornecer, a desenvolvedores de sistemas embarcados, uma técnica moderna para a validação de software, com a implementação de uma ferramenta para teste baseado em propriedades descritas utilizando lógica temporal e uma linguagem de domínio especifico embarcada imperativa para a programação segura nas linguagens correntes para o desenvolvimento destes sistemas. Para verificar a usabilidade da ferramenta desenvolvida, realizaram-se testes com escopos simples, próximos aos programadores, como o controle de dois semáforos sincronizados e a leitura de um sensor de temperatura, nos quais possíveis propriedades temporais foram especificadas e testadas, a fim de se observar a validação delas diante dos algoritmos testados.

Palavras-chaves: Lógica temporal. Teste de Software. Sistemas reativos.

Abstract

Tests are certainly the most commonly used approach by developers around the world for software quality assurance. However, building and running tests is an expensive activity, accounting for about 50 % of the cost of developing a system. Currently embedded systems are ubiquitous in modern society and are present in a variety of electronic devices where reliability and security are critical factors, such as banking applications and medical devices. Because its execution is determined by external events, testing embedded systems is a complex task, often done with the system in production, which can increase the cost of software development. In addition to the difficulties in constructing tests, reactive systems are not easily specified using formulas of the first-order logic, since they do not have mechanisms to represent, in a simple way, the temporal relation between actions of the system. Such formulations are made in a natural way using temporal logics, in which the notion of time is explicit in the semantics of formulas. The present work represents a technological contribution to provide, to developers of embedded systems, a modern technique for the validation of software, with the implementation of a test tool based on properties described using temporal logic and a specific domain language embedded imperative for the programming in the current languages for the development of these systems. To verify the usability of the developed tool, tests were performed with simple scopes, close to the programmers, as the control of two synchronized semaphores and the reading of a temperature sensor, in which possible temporal properties were specified and tested, in order to observe the validation of them before the algorithms tested.

Keywords: Temporal logic. Software testing. Reactive systems

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

EDSL	Embedded Domain Specific Language
CAGR	Compound Annual Growth Rate
LTL	Linear Temporal Logic
AST	Abstract Syntax Tree

Sumário

1 Introdução

A tecnologia é indiscutivelmente essencial e presente na vida de todos. Vemos cada vez mais sistemas computadorizados participando de processos a fim de otimizar tempo e precisão, entre outras variáveis, e diminuir custos. As aplicações são diversas e em muitas áreas, não só restrita a ambientes industriais, já em aparelhos comuns ao uso de muitos. A segurança e credibilidade dos sistemas são fundamentais para a seu completo funcionamento, e para garantir a ausência de erros e sua qualidade, os mesmos são submetidos a testes convencionais e simulações, todavia, a construção e execução destes são responsáveis por 50% do custo do desenvolvimento (MYERS, 2006). Tal custo motiva o estudo de técnicas para a automação completa ou parcial de testes. Ferramentas para construção e execução automática de testes permitem que esta tarefa seja feita em menor tempo, que um maior número de testes seja executado e que o processo de repetição de testes seja mais simples, assim que uma modificação seja realizada no software em questão.

Diversas técnicas para automação de testes foram propostas na literatura (MYERS, 2006), em especial o chamado teste baseado em propriedades, em que um software é verificado com respeito a uma propriedade expressa como uma fórmula da lógica. A implementação precursora desta abordagem é a biblioteca QuickCheck (CLAESSEN; HUGHES, 2011), implementada na linguagem Haskell (JONES, 2003b) e adaptada para outras linguagens de programação tais como Java, C++, Erlang, entre outras. Esta biblioteca é formada por duas linguagens de domínio específico embarcadas (EDSL[1]), a primeira para a descrição de propriedades as quais representam quantificadores e conectivos da lógica de primeira ordem, e a segunda para a criação de geradores de valores para testes, que juntas verificam se alguma propriedade especificada para valores aleatórios de entrada para uma função testada estão de acordo com o seu resultado.

1.1 Problema

Hoje em dia, sistemas embarcados é uma das tecnologias inerentes à rotina de muitas pessoas como citado inicialmente. É exigido alto grau de confiabilidade desses sistemas e os exemplos são diversos com diferentes níveis de complexidade, como de controle de intertravamentos de metroviários, controle automático de aeronaves e controles de usinas nucleares, aplicações onde a corretude e segurança são críticas quanto a performance, tal que uma simples falha possa colocar vidas humanas em risco tão quanto o meio ambiente

[1] Abreviação do inglês, Embedded Domain Specific Language. Dizemos que uma linguagem de domínio específica é embarcada se esta é definida utilizando construções de uma outra linguagem, isto é, são implementadas como bibliotecas.

(FERREIRA, 2005); ou até mesmo dispositivos eletrônicos como celulares, aparelhos GPS, eletrodomésticos e vários outros os quais estamos habituados a usar nas diversas tarefas cotidianas.

Sistemas embarcados são normalmente implementados como sistemas reativos, ou seja, programas cuja execução é acionada mediante a eventos externos do ambiente como, por exemplo, a leitura de um sensor de temperatura, infravermelho, ultrassônico, etc. Testar tais sistemas é uma tarefa complexa sendo muitas vezes feita com o sistema em produção, o que pode resultar no aumento do custo do desenvolvimento do software. Além das dificuldades para construção de testes, sistemas reativos não são facilmente especificados utilizando fórmulas da lógica de primeira ordem, uma vez que esta não possui mecanismos para representar, de forma simples, a relação temporal entre ações do sistema (BAIER; KATOEN, 2008; CLARKE; GRUMBERG; PELED, 1999).

As verificações de sistemas reativos são realizadas satisfatoriamente com o uso de verificadores de modelos (model checkers), ferramentas como o NuSMV e o SPIN (HOLZMANN, 2003), as quais modelam os sistemas como máquinas de estados (BÖRGER; STÄRK, 2012) nas quais são verificadas propriedades como fórmulas da lógica temporal (PNUELI, 1977). Ao se tratar de validação de softwares, o uso destes verificadores de modelos é impraticável dado ao grande número de estados gerados até mesmo para algoritmos de médio porte.

1.2 Objetivos

Este estudo visa alcançar o objetivo geral e os objetivos específicos, relacionados a seguir.

1.2.1 Objetivo geral

Especificar e implementar uma biblioteca para teste baseado em propriedades para sistemas embarcados, utilizando lógica temporal para especificação de propriedades destes sistemas.

1.2.2 Objetivos específicos

- Analisar o estado da arte em teste baseado em propriedades e teste de sistemas embarcados;
- Projetar e implementar uma ferramenta de verificação em Haskell para especificação de propriedades expressas usando lógica temporal;

- Projetar e implementar uma linguagem de domínio específico embarcada imperativa para o desenvolvimento de dispositivos embarcados.
- Projetar e implementar uma biblioteca de teste baseado em propriedades (descritas usando a linguagem implementada no segundo objetivo acima), usando a linguagem Haskell, para criação de entradas aleatórias para teste;
- Validar a ferramenta desenvolvida para implementação e teste de sistemas embarcados, especificamente na plataforma Arduíno.

1.3 Justificativa

O mercado de sistemas embarcados possui uma expectativa de taxa de crescimento significativa entre 2017 e 2023, o qual é esperado que cresça a uma CAGR[2] de 4,05% entre este intervalo e deverá ser avaliado em US $ 110,46 bilhões até 2023. O mercado vem sendo impulsionado por diversos fatores de aumento na adoção dos sistemas embarcados como na indústria automotiva, uso de tecnologia de processador multicore em aplicações militares, crescente uso de dispositivos móveis e aplicações IoT, aumento da demanda em equipamentos de saúde e por fim o fator mais crítico pelo aumento deste mercado, a preocupação com a segurança (MARKETS; MARKETS, 2017). A diversidade de áreas das quais os sistemas embarcados participam motiva as atividades de pesquisa e desenvolvimento (P&D) que com o decorrer do tempo inovam-se em aplicações cada vez mais eficientes a nível de complexidade, confiabilidade e custo.

O suporte para testes adequado para o desenvolvimento de sistemas embarcados pode reduzir drasticamente o custo de produção e consequentemente, aumentar sua qualidade. É conhecido que grande parte do esforço de um desenvolvimento de um software está em etapas de validação (SOMMERVILLE, 1995). Com isso, usualmente, empresas de desenvolvimento valem-se apenas de testes não automatizados ou de estratégias ineficazes para a automação destes. Nesse sentido, o presente trabalho representa uma contribuição tecnológica por fornecer, a desenvolvedores de sistemas embarcados, técnicas modernas para a validação de software, uma vez que, pelo melhor do nosso conhecimento, não há solução para a especificação e teste baseado em propriedades da lógica temporal em tais sistemas.

[2] Abreviação do inglês, Compound Annual Growth Rate. Índice que representa a taxa de retorno de um investimento.

1.4 Estrutura do trabalho

Este trabalho está organizado de acordo os seguintes capítulos:

O capítulo 2 possui o estado da arte da verificação de sistemas reativos e também um referencial teórico de conceitos e técnicas utilizadas neste trabalho.

No capítulo 3 é descrito uma visão geral da ferramenta construída, assim como todos os módulos que a constitui.

O capítulo 4 é dedicado as descrições dos testes realizados e constatação da usabilidade do verificador construído.

Por fim, o capítulo 5 traz uma conclusão do trabalho e também sugestões de aprimoramento da ferramenta como trabalhos futuros.

2 Revisão bibliográfica

2.1 Lógica Temporal

Como descrito no Capítulo 1, sistemas reativos não são facilmente especificados utilizando fórmulas da lógica de primeira ordem, dado suas execuções serem determinadas por eventos externos e tal lógica não possuir propriedades que forneçam uma modelagem adequada. Em vista disso, a corretude destes sistemas é dependente das suas execuções, não somente das suas entradas e saídas, aspectos dos quais o formalismo da lógica temporal consegue tratar ao estender a lógica proposicional ou de predicados por modalidades permitindo referenciar o comportamento infinito de sistemas reativos (BAIER; KATOEN, 2008). A lógica temporal é dividida em duas categorias das quais se diferenciam no modo como o tempo é caracterizado, de forma linear ou ramificado. No primeiro, a cada momento no tempo há somente um sucessor, enquanto no segundo existe uma ramificação na qual o tempo se divide em cursos alternativos.

Algoritmos implementados para sistemas embarcados possuem uma estrutura sequencial de controle, isto é, os comandos em um programa ou função são executados um após o outro na ordem em que foram especificados, de acordo com um modelo do qual há entradas de dados, algum tipo de processamento e por fim uma saída de dados. Diante disso, uma abordagem baseada em uma perspectiva linear sobre o tempo é a mais adequada, a chamada LTL [1] cujo domínio de tempo é discreto, ou seja, o presente momento refere-se ao estado atual e o próximo momento corresponde ao imediato estado sucessor (BAIER; KATOEN, 2008).

2.1.1 Lógica Temporal Linear

A lógica temporal linear, assim como a lógica de primeira ordem, possui conectivos e operadores em sua sintaxe, assim como suas semânticas correspondentes. Existem diversas formas na literatura de representações dos conectivos temporais, uma delas, que fundamenta este trabalho, é a dos autores do livro *Principles of Model Checking* que estão presentes nas seguintes seções.

2.1.1.1 Sintaxe e semântica

Os elementos básicos de uma fórmula LTL são proposições atômicas, conectivos booleanos como o de conjunção $\wedge$ e negação $\neg$ e dois conectivos temporais básicos, o conectivo $\bigcirc$ (pronunciado como "next") e U (pronunciado como "until"), sendo o primeiro

[1] Abreviação do inglês, Linear Temporal Logic.

um operador unário e o segundo binário. As fórmulas LTL sobre um conjunto AP de proposições atômicas são formadas de acordo com a gramática:

$$\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \text{ U } \varphi_2$$

onde $a \in AP$.

Utilizando os conectivos booleanos $\wedge$ e $\neg$, podemos obter outros como a disjunção $\vee$, implicação $\rightarrow$, equivalência $\leftrightarrow$ e o operador ou exclusivo $\oplus$, de acordo com as seguintes equivalências:

$$\begin{array}{lcll} \varphi_1 & \vee & \varphi_2 & \equiv \quad \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 & \rightarrow & \varphi_2 & \equiv \quad \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 & \leftrightarrow & \varphi_2 & \equiv \quad (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\ \varphi_1 & \oplus & \varphi_2 & \equiv \quad (\varphi_1 \wedge \neg\varphi_2) \vee (\varphi_2 \wedge \neg\varphi_1) \end{array}$$

Podemos também derivar outros conectivos temporais a partir do conectivo temporal U, como o $\Diamond$ (pronunciado como "eventually") e o $\Box$ (pronunciado como "always"), de acordo com as seguintes equivalências:

$$\begin{array}{lcl} \Diamond\varphi & \equiv & \text{true U } \varphi \\ \Box\varphi & \equiv & \neg\Diamond\neg\varphi \end{array}$$

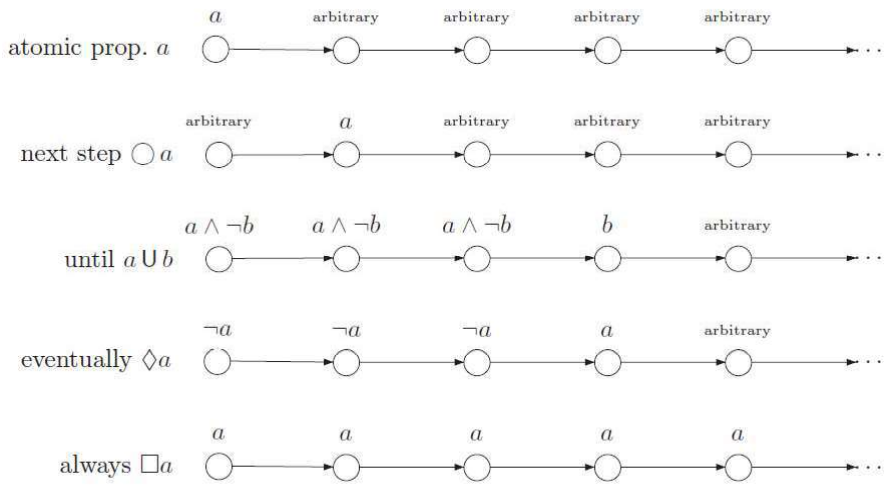
A semântica dos operandos temporais é definida a partir de uma trajetória específica de uma sequência de estados. Assim, os seguintes conectivos possuem os significados:

1. $\bigcirc\varphi$ - φ será válida no próximo estado.
2. $\Diamond\varphi$ - φ será válida eventualmente no futuro.
3. $\Box\varphi$ - φ é válida e sempre será válida.
4. φ_1 U φ_2 - φ_1 é válida até que φ_2 seja válida.

Um outro conectivo binário bastante útil é o W (pronunciado como "unless"), também conhecido como *until* fraco, cuja semântica é semelhante a do operador *until* diferindo apenas na exigência feita pelo U de que a fórmula à sua direita seja válida em algum instante futuro e W não exige esta condição. O conectivo W possui a seguinte definição formada com o uso dos outros conectivos temporais já conhecidos:

$$\varphi_1 \text{ W } \varphi_2 \equiv \Box\varphi_1 \vee (\varphi_1 \text{ U } \varphi_2)$$

Figura 1 – Semântica intuitiva dos operadores temporais



Fonte: (BAIER; KATOEN, 2008)

2.2 Verificadores de modelo

Uma destas técnicas apropriadamente utilizada na verificação de sistemas reativos é a chamada verificação de modelos (amplamente conhecida como *model checking*), que através de um algoritmo de bruta força, varre todos os possíveis estados de um sistema, examinando todos os possíveis cenários de maneira sistemática. A implementação pioneira deste verificador foi feita nos anos 80 (CLARKE; EMERSON; SISTLA, 1986), quando uma alternativa técnica de verificação chamada *temporal logic model checking* foi desenvolvida por Clarke e Emerson nos Estados Unidos de grande valor para áreas da ciência da computação, particularmente no projeto de circuitos digitais e protocolos de comunicação (CLARKE; GRUMBERG; PELED, 1999).

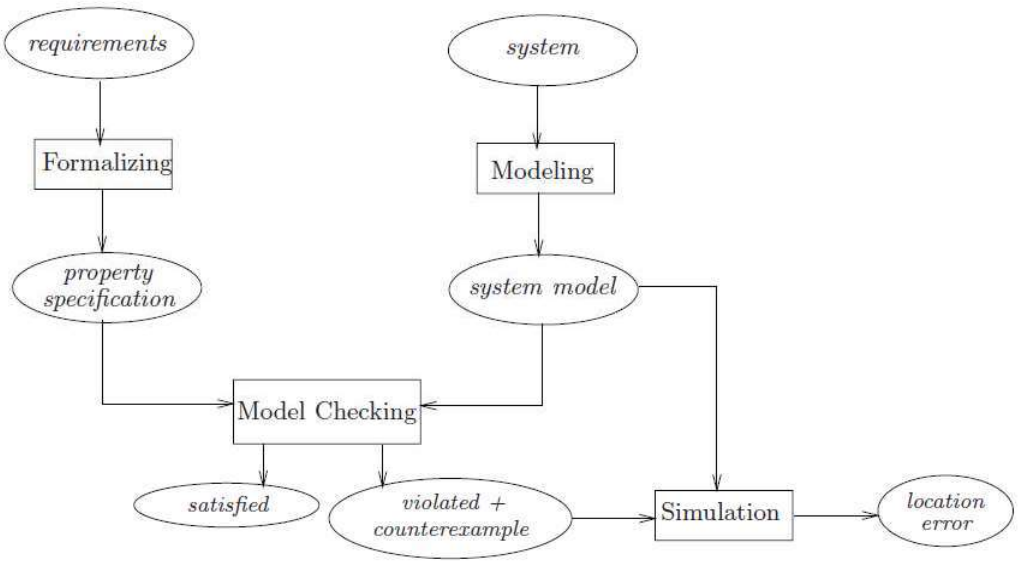
Segundo BAIER; KATOEN o princípio de um verificador de modelo é definido como:

> Model checking é uma técnica automatizada que, dado um modelo de estado finito de um sistema e uma propriedade formal, é verificado sistematicamente se essa propriedade é válida para um estado n neste modelo.

Quando o verificador encontra um estado que violou a propriedade especificada inicialmente, ele fornece um contraexemplo, que indica a forma como o modelo atingiu tal estado indesejado, o qual descreve toda a trajetória de execução, desde o estado inicial do sistema ao estado comprometido.

Aplicar o verificador em um projeto, consiste em diversas etapas as quais agrupam-se em três fases (CLARKE; GRUMBERG; PELED, 1999):

Figura 2 – Vista esquemática da abordagem Model checking



Fonte: (BAIER; KATOEN, 2008)

1. **Modelagem** - conversão do projeto em um formalismo aceitável pelo verificador. Em diversos casos, o modelo é uma abstração da qual detalhes irrelevantes são eliminados devido a limitações de memória e tempo.

2. **Especificação -** descrever o conjunto de propriedades a serem testadas as quais o sistema deve satisfazer. Normalmente para sistemas de software e hardware utiliza-se a lógica temporal que pode certificar como o comportamento do sistema evolui ao longo do tempo.

3. **Verificação** - realizada automaticamente, verifica as propriedades para os estados e na ocorrência de insatisfatibilidade, o retorno de um contraexemplo é dado como informação necessária à solução do problema, o qual pode ser uma modelagem incorreta do sistema ou uma especificação incorreta.

A principal desvantagem do verificador é a explosão de estados que pode ocorrer diante de um sistema com muitos componentes, ou seja, o número de estados necessários para modelar o sistema com precisão pode facilmente exceder a quantidade de memória disponível. Logo seu uso é inviável para softwares de médio a grande porte que usualmente fazem parte de sistemas mais complexos e que exigem maior confiabilidade e segurança.

2.3 Linguagem de domínio específico

Uma abordagem genérica fornece uma solução geral para uma série de problemas em uma certa área, todavia muitas vezes tal solução é subotimizada, isto é, nem sempre

o resultado provém de uma completa análise do problema dentro de um escopo. Uma abordagem específica provê uma melhor solução para um conjunto pequeno de problemas que possuem um nicho particular de aplicação, mais especializado. Uma linguagem de domínio específico (DSL[2]) utiliza esta última abordagem. É uma linguagem pequena, usualmente declarativa que oferece um expressivo poder, restrita ao domínio do problema (DEURSEN; KLINT; VISSER, 2000).

Dentre as principais motivações para se utilizar uma DSL estão o ganho de produtividade, confiabilidade e portabilidade ao se especificar uma linguagem próxima ao domínio do seu utilizador, maior facilidade de compreensão do código e possibilidade de uso por uma pessoa que não possui um profundo entendimento de programação de computadores. Estas vantagens refletem em diversas linguagens amplamente conhecidas e utilizadas em diversas aplicações, das quais as mais clássicas estão PIC, CHEM, YACC e Make (BENTLEY, 1986), além de outras mais conhecidas como SQL, HTML, CSS e BNF. Esta diversidade de linguagens é consequência da variedade de domínios que podem ser agrupados em áreas como engenharia de software, telecomunicações, sistemas multimídia, sistemas de software e outras como controle robótico e solucionadores de equações diferenciais parciais. A metodologia de projeto de uma DSL tradicionalmente envolve 3 fases que compreende etapas de seu desenvolvimento (DEURSEN; KLINT; VISSER, 2000).

1. Análise - identificação do domínio do problema assim como todas informações necessárias, a fim de se obter um conjunto de informações semânticas que o modele.

2. Implementação - construção de uma biblioteca que implemente suas semânticas. Projeto e implementação de um compilador ou interpretador para a linguagem que traduza programas DSL para uma sequência de chamadas de biblioteca.

3. Utilização - escrita dos programas DSL e sua compilação/interpretação que obtenha os resultados desejados para as aplicações.

Neste trabalho foram construídas EDSL's que são linguagens definidas em termos de uma linguagem hospedeira, ou seja, implementadas como biblioteca possuindo a vantagem de se utilizar da infra-estrutura de uma outra linguagem, como *parsing*, *typechecking*, etc. Como mencionado na subseção 1.2.2, utilizamos a linguagem funcional Haskell para as construções das novas bibliotecas, assim como toda programação realizada.

2.4 Teste baseado em propriedades

Técnicas de automatização de testes fazem parte de uma área em potencial expansão que aos poucos vem se desenvolvendo. Muitos dos projetos de automatização de testes são

[2] Usualmente referenciado como DSL, sigla da tradução em inglês

resultados de processos empíricos de tentativa e erro, o que motiva a construção de novas alternativas que buscam uma abordagem diferente a fim de se obter uma melhor automação. Como supracitado, diversas técnicas de automatização de testes foram propostos na literatura, em especial o teste baseado em propriedades, cuja implementação pioneira desta abordagem é a biblioteca QuickCheck, um dos principais trabalhos relacionados ao presente trabalho.

A biblioteca QuickCheck inicialmente foi projetada na linguagem Haskell por Claessen e Hughes (CLAESSEN; HUGHES, 2011) e posteriormente portada para outras linguagens de diferentes paradigmas de programação. Ela é formada por duas linguagens de domínio específico embarcada, sendo uma delas utilizada para a especificação de propriedades, em que cada construção representa quantificadores e conectivos da lógica de primeira ordem e uma para criação de geradores de valores para testes, com base no tipo destes valores, de forma que o QuickCheck possa verificar as propriedades para uma larga escala de casos.

As propriedades são apenas funções simples que retornam um valor booleano, verdadeiro ou falso, para argumentos criados aleatoriamente pela biblioteca de geradores de valores para testes que podem ser definidos pelos programadores com controle sobre a distribuição de dados de teste. Caso o resultado seja verdadeiro para todos os casos testados, ele reporta "OK"e a quantidade de testes que foram avaliados (valor arbitrário), contudo quando um caso de teste que invalida alguma propriedade é encontrado, o QuickCheck tenta reduzi-lo a um subconjunto mínimo de falhas, removendo ou simplificando os dados de entrada que não são necessários para fazer o teste falhar (CLAESSEN; HUGHES, 2011).

O presente trabalho possui uma forte relação com esta biblioteca utilizada para softwares convencionais. O intuito é de se criar uma ferramenta de verificação à maneira da QuickCheck, todavia para softwares portados em sistemas reativos utilizando, ao invés da lógica de primeira ordem, a lógica temporal.

3 Metodologia

3.1 Visão geral

A implementação deste trabalho foi dividida em etapas com as quais desenvolvemos sequencialmente conforme foram necessárias. A linguagem escolhida para toda a codificação foi a de paradigma funcional Haskell, uma programação cuja operação fundamental é a aplicação de funções aos argumentos. Normalmente, definimos a função principal em termos de outras funções, que por sua vez são definidas em termos de ainda mais funções. O paradigma de programação funcional, em contraste com a imperativa, não possui construções como declarações de variáveis ou instruções de atribuições, por isso, uma vez dado um valor, ele nunca muda. Dentre as várias vantagens, esta linguagem nos fornece toda uma estrutura adequada ao tipo do problema tratado, ao não permitir efeitos colaterais em funções [1], oferecendo uma transparência referencial, a qual assegura que o resultado de uma função será o mesmo para um dado conjunto de parâmetros não importando onde ou quando seja avaliada, o que a caracteriza como uma função pura (HUGHES, 1989). Funções puras são muito mais fáceis de testar visto que não é necessário se preocupar com um estado antes e depois da execução, portanto, são ideais para implementação dos módulos que compõem este trabalho.

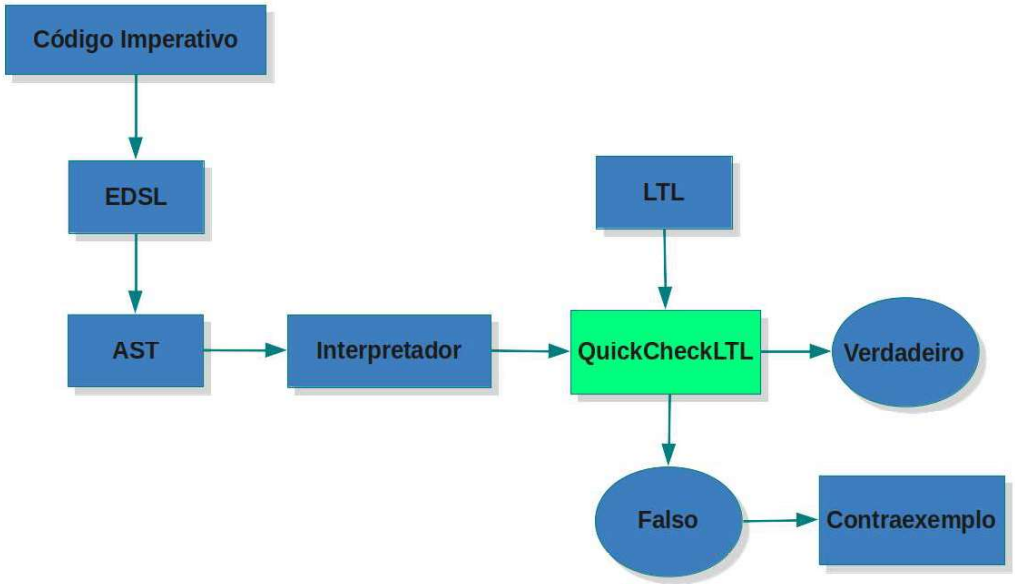
De modo a cumprir os objetivos específicos propostos na subseção 1.2.2, inicialmente pesquisamos sobre as atuais técnicas de verificação de sistemas reativos a fim de se entender as diferentes abordagens realizadas e quais são suas limitações, para então atentar sobre os diferencias que este trabalho vem a propor. Programas para sistemas reativos são normalmente codificados utilizando o paradigma imperativo e é a partir deste conhecimento que partimos para o desenvolvimento de uma EDSL imperativa. De modo geral, criamos alguns elementos que usualmente fazem parte do alfabeto de uma linguagem imperativa tais como laços de repetição, operadores aritméticos e lógicos, declarações e atribuições de variáveis de diferentes tipos, entre outros. Uma vez definidas estas construções, podemos montar árvores de sintaxe abstratas (ASTs) que representam estruturas sintáticas de cadeias como por exemplo expressões matemáticas, que inserimos como entrada para o nosso interpretador da EDSL criada. Logo após esta etapa, temos a implementação de uma EDSL para especificação de propriedades expressas usando lógica temporal linear (LTL) para que possamos utilizá-la em um verificador de programas interpretados da nossa EDSL imperativa que retorna verdadeiro se todas as propriedades descritas foram atendidas, ou retorna falso, na qual pelo menos uma propriedade foi invalidada, e logo em

[1] Um função é dita ter efeito colateral se modificar algum estado fora de seu escopo, como por exemplo funções de E/S

seguida um contraexemplo.

Esta rápida descrição do presente trabalho é ilustrada na figura 3 abaixo, na qual podemos observar como os módulos se relacionam e a ordem em que a verificação de um programa segue. O módulo QuickCheckLTL é onde todas as partes do trabalho por completo são agregadas e é nele em que definidos funções que controlam os demais módulos, bem como quais as propriedades LTL que serão verificadas para um dado código de entrada. As próximas seções deste capítulo contém uma descrição mais detalhada do desenvolvimento de cada módulo do presente trabalho.

Figura 3 – Vista esquemática da solução



Fonte: Elaborada pelo autor.

3.2 Módulo AST

Árvores de sintaxe abstrata são representações de estruturas sintáticas abstratas de códigos escritos em alguma linguagem de programação que, diferentemente de árvores concretas, se abstraem de alguns detalhes para se concentrar apenas na estrutura sintática, como por exemplo a omissão de nós de árvore para representar sinais de pontuação como ponto-e-vírgula para finalizar instruções ou vírgulas para separar argumentos de uma função (JONES, 2003a).

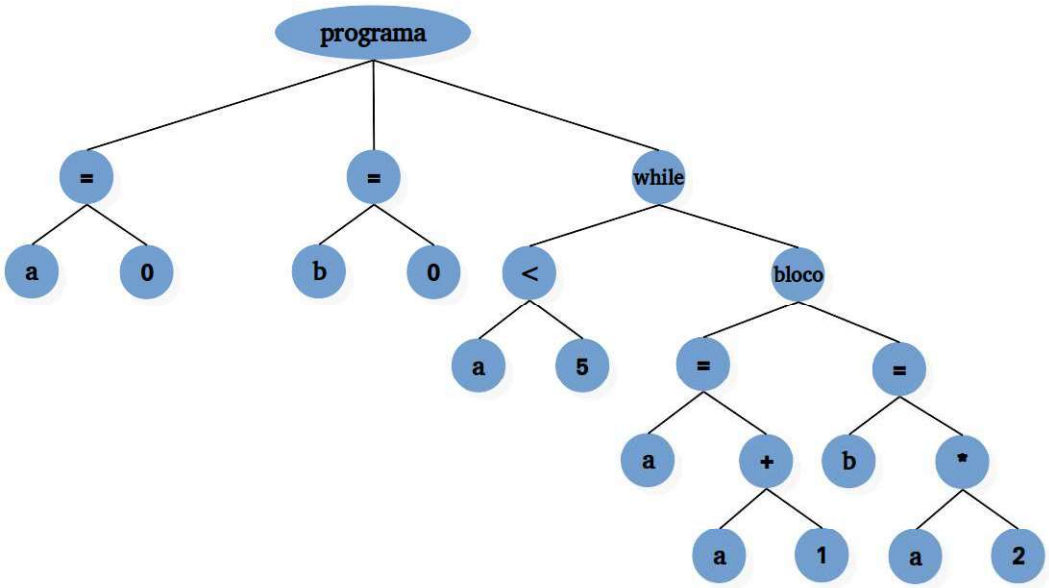
Com o intuito de exemplificar a montagem de uma árvore dado um código imperativo, de acordo com cada identificador, operador e outros elementos do trecho de código da figura 4, a árvore gerada é a ilustrada na figura 5, na qual podemos observar a relação entre os seus elementos e de que forma as construções são montadas, como no laço de repetição while.

Figura 4 – Trecho de código imperativo

```
int a = 0;
int b = 0;
while (a < 5)
{
   a = a + 1;
   b = a * 2;
}
```

Fonte: Elaborada pelo autor.

Figura 5 – Exemplo de uma AST



Fonte: Elaborada pelo autor.

A fim de se representar um programa imperativo, neste módulo foram criados novos tipos de dados comumente feitos em Haskell utilizando a palavra chave **data** para definir um novo tipo, como por exemplo o tipo Bool já definido na biblioteca padrão:

```
data Bool = False | True
```

A parte a esquerda do sinal (=) diz o nome, que no caso é o Bool e a da direita são os construtores de valores, ou seja, os diferentes valores que o tipo pode assumir, False ou True (LIPOVACA, 2011).

Os novos tipos de dados criados foram os Type, Stmt, Exp e Value, os quais resumidamente representam, respectivamente, domínios de tipos (e.g., inteiro, booleano, char), declarações (e.g., atribuição, laço de repetição, condicionais), expressões (e.g., operadores aritméticos, lógicos, booleanos, literais, variável) e valores (e.g., inteiros, booleano, ponto

flutuante). É a partir dos construtores que conseguimos definir as notações de cada novo tipo criado e consequentemente podemos definir quais são operadores binários, como soma e divisão, e operadores unários como o operador lógico de negação, no tipo Exp, bem como no tipo Stmt no qual definimos, por exemplo, que uma atribuição é composta por um nome (string) e uma expressão (Exp), que pode ser um valor literal, uma outra variável ou até mesmo uma operação aritmética como acontece nas atribuições das variáveis "a"e "b"da figura 4.

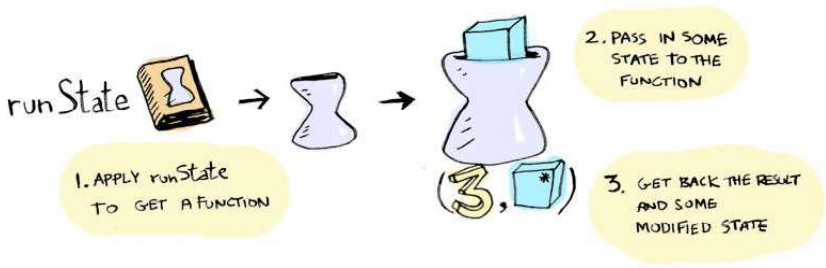
3.3 Módulo EDSL imperativa

O módulo EDSL imperativa possui as funções que irão montar as ASTs para os respectivos trechos de códigos. Para cada construtor dos tipos Exp e Stmt foi criada a função que montam a sua respectiva árvore. Neste módulo é utilizado a mônada de estados que pertence a biblioteca Control.Monad.State, que possui a definição:

```
newtype State s a = State { runState :: s -> (a,s) }
```

no qual um *Statesa* é uma computação de estado que manipula um estado do tipo s e tem um resultado do tipo a (LIPOVACA, 2011). Uma mônada de estado encapsula uma função que recebe um estado e retorna um valor e um novo estado.

Figura 6 – Funcionamento de uma mônada de estado



Fonte: Adit's homepage [2].

A mônada de estados nos permite simular uma execução sequencial das linhas do código imperativo, de modo que a cada linha é gerado um novo estado adicionando um novo comando desta linha, isto é, uma nova declaração (atribuição, declaração, etc.), e também uma adição de uma variável e seu respectivo Type em um mapeamento, caso o comando seja de declaração de variável.

A função de atribuição desta linguagem verifica se a variável já faz parte do ambiente de variáveis e se sim, a inclui em uma lista de Stmt. Temos também funções para a utilização de condicionais (if-then e if-then-else) e laço de repetição (while).

[2] Disponível em: <http://adit.io/posts/2013-06-10-three-useful-monads.html> Acesso em jul. 2018.

Este módulo também possui uma verificação de tipos das expressões (Exp) no qual é averiguado se são passados tipos iguais para operações aritméticas assim como para as operações lógicas.

De maneira a exemplificar seu funcionamento, podemos considerar o trecho de código da figura 4 e montá-lo utilizando as funções deste módulo:

Figura 7 – Código escrito com a EDSL

```
exemplo = do
            declare "a" TyInt (int 0)
            declare "b" TyInt (int 0)
            while ((var "a") .<. (int 5))
                  (do
                     assign "a" ((var "a") .+. (int 1))
                     assign "b" ((var "a") .*. (int 2))
                  )
```

Fonte: Elaborada pelo autor.

como as funções de atribuição (assign), a de representação de variável (var), o laço de repetição (while) e as de operações matemáticas. A notação **do** nos permite encadear sequências de comandos EDSL e assim, propagar um estado de uma declaração para o próximo. A EDSL gera como resultado a seguinte AST:

Figura 8 – AST gerada

```
[Declare "a" (ILit 0),
 Declare "b" (ILit 0),
 While (Lt (EVar "a") (ILit 5))
       [Assign "a" (Plus (EVar "a") (ILit 1)),
        Assign "b" (Times (EVar "a") (ILit 2))]]
```

Fonte: Elaborada pelo autor.

conforme os construtores definidos no tipo Stmt [(Assign String Exp) e (While Exp [Stmt])] e no tipo Exp [(EVar String), (Plus Exp Exp), (Times Exp Exp) e (Lt Exp Exp)], ambos pertencentes ao AST. O resultado das funções aplicadas acima é uma lista de declarações (Stmt) que representa a árvore montada para o código escrito, juntamente com o seu ambiente de variáveis.

Funções características de sistemas reativos, como função de delay e de configuração, leitura e escrita de pinos, também foram adicionados nesta EDSL. A função de configuração de pino determina quais pinos serão utilizados como entrada ou saída de dados. A de leitura de pino armazena um valor lido do pino em uma variável existente. Já a de escrita

de um pino escreve um valor em um pino especificado de saída. Também temos a função de delay a qual incrementa o tempo de execução do estado com o valor em milissegundos passado para ela.

De maneira a organizar a transformação dos códigos de entrada, o módulo CodeLibrary é onde escrevemos os código escritos com a linguagem EDSL imperativa, com base nos códigos-fonte.

3.4 Módulo Interpretador

A implementação do interpretador para a linguagem EDSL imperativa é necessária para traduzir programas EDSL para uma sequência de chamadas de biblioteca. Neste módulo também utilizamos a mônada de estados, na qual um estado s possui um mapeamento de nomes de variáveis e seus respectivos valores (tipo Value da AST), e um valor do tipo *integer* que representa o tempo de execução. O interpretador da nossa AST é formado basicamente por duas principais funções, que dado a AST (lista de declarações), criada pelo módulo anterior, irão respectivamente avaliar expressões (Exp) e retornar um valor (Value), e executar as declarações uma a uma retornando sempre um novo estado, criando todo o traço do programa até o momento.

A semântica da AST gerada pela EDSL é descrita pelas funções semânticas $\mathbb{E}$ e $\mathbb{S}$. A função $\mathbb{E}(e, s) \rightsquigarrow (s', y)$ mapeia uma AST de expressão e e um estado s para um par (s', y), onde s' é um novo estado e y um valor. A função semântica $\mathbb{S}(d, s) \rightsquigarrow s'$ mapeia uma AST de um declaração d e um estado s para um novo estado s'. Para representar mudanças no estado das variáveis no mapeamento usamos a notação $(m\{v \rightarrow y\})$, que significa que criamos uma cópia do estado corrente com todos os seus valores, adicionando o mapeamento da variável v para o valor y, ou então sobrescrevendo-o caso ele já exista. Para representar que acessamos o conteúdo de uma variável usamos a notação $m(x)$ que significa o valor da variável x no mapeamento m. A semântica da função $\mathbb{E}$ é dada na figura 9.

Figura 9 – Semântica de expressões

$$
\begin{array}{lcl}
\mathbb{E}\ (val\ n)(\mathrm{m,t}) & = & ((m,t),n) \\
\mathbb{E}\ (var\ x)(\mathrm{m,t}) & = & ((m,t'),m(x)) \\
\mathbb{E}\ (e1 \oplus e2)(\mathrm{m,t}) & = & \mathbb{E}\ (e1,(m,t)) \rightsquigarrow ((m,t'),v1) \\
 & & \mathbb{E}\ (e2,(m,t')) \rightsquigarrow ((m,t''),v2) \\
 & & ((m,t''),v1 \oplus v2)
\end{array}
$$

Fonte: Elaborada pelo autor.

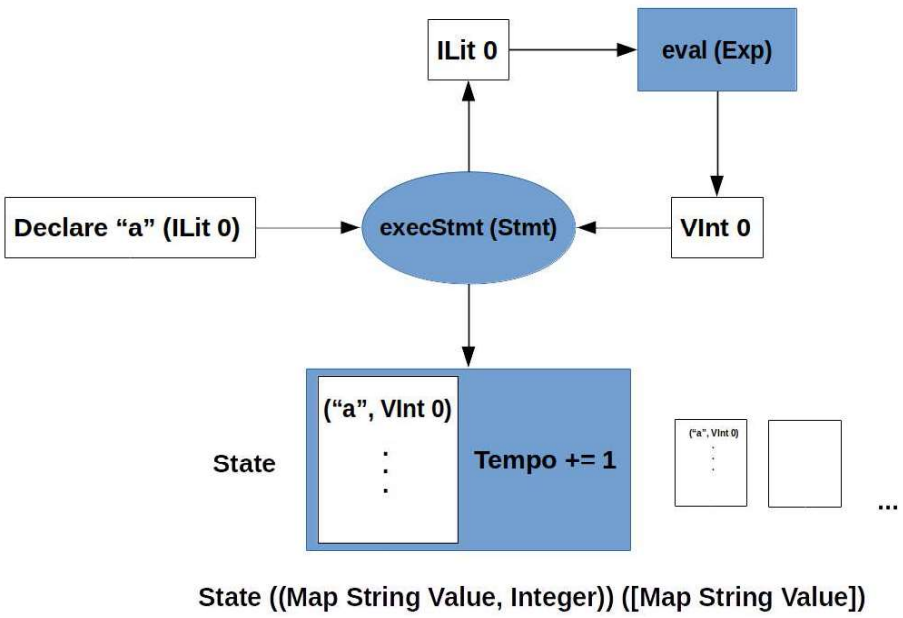
A semântica de execução para cada declaração é definida como na figura 10. A implementação em Haskell é praticamente uma tradução direta das funções semânticas $\mathbb{S}$ e $\mathbb{E}$.

Figura 10 – Semântica de declarações

$$
\begin{aligned}
\mathbb{S}\ [\text{Declare s e}]\text{:ys (m,t)} &= \mathbb{E}\ (e,(m,t)) \rightsquigarrow ((m,t'),v) \\
&\quad\ \mathbb{S}\ [ys]\ (m'\{s \rightarrow v\},t') \\
\\
\mathbb{S}\ [\text{Assign s e}]\text{:ys (m,t)} &= \mathbb{E}\ (e,(m,t)) \rightsquigarrow ((m,t'),v) \\
&\quad\ \mathbb{S}\ [ys]\ (m'\{s \rightarrow v\},t') \\
\\
\mathbb{S}\ [\text{If e xs}]\text{:ys (m,t)} &= \text{Se } (\mathbb{E}\ (e,(m,t))) \rightsquigarrow ((m,t'),True)) \\
&\quad\ \text{Então } \mathbb{S}\ [xs]\ (m,t) \rightsquigarrow (m',t'') \\
&\qquad\ \mathbb{S}\ [ys]\ (m',t'') \\
&\quad\ \text{Senão } \mathbb{S}\ [ys]\ (m',t'') \\
\\
\mathbb{S}\ [\text{While e xs}]\text{:ys (m,t)} &= \text{Se } (\mathbb{E}\ (e,(m,t))) \rightsquigarrow ((m,t'),True)) \\
&\quad\ \text{Então } \mathbb{S}\ [xs]\ (m,t') \rightsquigarrow (m',t'') \\
&\qquad\ \mathbb{S}\ [\text{While e xs}]\text{:ys (m',t'')} \\
&\quad\ \text{Senão } \mathbb{S}\ [ys]\ (m',t')
\end{aligned}
$$

Fonte: Elaborada pelo autor.

Figura 11 – Exemplo de execução de uma declaração de variável



Fonte: Elaborada pelo autor.

A figura 11 ilustra a execução de uma declaração de variável, a qual possui uma expressão (ILit 0) que é dada como entrada para o avaliador de expressões e este retornará

um valor (VInt 0). Conforme novas variáveis são criadas ou modificadas, os estados são criados contendo o ambiente de variáveis alterado e o tempo de execução incrementado a partir da padronização da tabela 1 .

Tabela 1 – Tempo de execução de comandos

Comandos	Tempo de execução (μs)
Literal	zero
Variável	1 (tempo de buscar seu valor atual)
Declaração	tempo de avaliar expressão + 1
Atribuição	tempo de buscar o valor atual da variável + avaliar expressão + 1
If-then	tempo de avaliar expressão + tempo somado do bloco then
if-then-else	tempo de avaliar expressão + tempo somado do bloco then + else
While	tempo de avaliar expressão + tempo somado das repetições do bloco
Operador binário	tempo de avaliar lado direito e esquerdo do operador + 1
Operador unário	tempo de avaliar a expressão + 1

Fonte: Elaborada pelo autor.

A saída do interpretador para o código EDSL imperativa escrito para o mesmo exemplo das seções anteriores será:

Figura 12 – Saída do interpretador

```
[fromList [("a",VInt 0)],
 fromList [("a",VInt 0),("b",VInt 0)],
 fromList [("a",VInt 1),("b",VInt 0)],
 fromList [("a",VInt 1),("b",VInt 2)],
 fromList [("a",VInt 2),("b",VInt 2)],
 fromList [("a",VInt 2),("b",VInt 4)],
 fromList [("a",VInt 3),("b",VInt 4)],
 fromList [("a",VInt 3),("b",VInt 6)],
 fromList [("a",VInt 4),("b",VInt 6)],
 fromList [("a",VInt 4),("b",VInt 8)],
 fromList [("a",VInt 5),("b",VInt 8)],
 fromList [("a",VInt 5),("b",VInt 10)],
 fromList [("a",VInt 5),("b",VInt 10)]]
```

Fonte: Elaborada pelo autor.

onde vemos a sequência de estados com as mudanças realizadas nas variáveis "a"e "b".

3.5 Módulo LTL

Neste módulo implementamos um verificador de propriedades da lógica temporal linear. De acordo com os conectivos que a LTL possui, criamos um novo tipo de dado, da

mesma forma da seção 3.2, contendo os construtores de cada operador lógico e temporal, cuja sintaxe e semântica foram explicados na seção 2.1.

Figura 13 – Novo tipo de dado - LTL

```
data LTL = Atom String
         | TT
         | FF
         | LTL :&: LTL
         | LTL :|: LTL
         | Not LTL
         | G LTL -- always
         | F LTL -- eventually
         | X LTL -- next
         | U LTL LTL -- until
         | W LTL LTL -- unless
```

Fonte: Elaborada pelo autor.

Uma vez realizado todo o traçado do código de entrada em uma lista de mapeamentos de variáveis e valores, resultado do interpretador que simula a execução do algoritmo, podemos verificar a satisfatibilidade de certas propriedades temporais para o escopo com o qual o código de entrada está inserido. Isto é realizado através da função recursiva checkLTL, que simplesmente implementa a semântica dos operadores para uma sequência de estados. O resultado desta função é um par contendo um valor booleano e um mapeamento de variáveis, ou seja, verdadeiro ou falso para as propriedades especificadas no código e um contraexemplo, somente se pelo menos uma propriedade não foi validada.

A figura 14 possui a implementação da função checkLTL. A função irá verificar se dado uma condição a ser verificada e uma fórmula da LTL, cada elemento da sequência de estados es atende tal propriedade. Para o conectivo TT, o verificador apenas retorna o par $(True, Nothing)$ que representa a satisfatibilidade da propriedade verificada e por isso não retorna um contraexemplo, representado pelo $Nothing$. Já para o conectivo FF, o retorno é o par $(False, c)$, onde $False$ significa a não validação da propriedade e para tal, o primeiro estado c invalidado é dado como contraexemplo. Para uma fórmula atômica apenas verificamos se o átomo corresponde a condição passada dentro do atual estado verificado. Para os conectivos $\wedge$ e $\vee$, aplicamos o verificador para ambos os lados da fórmula LTL, e para o $\neg$, negamos o resultado da verificação.

O conectivo G foi implementado utilizando a função lstCheck que verifica se a propriedade definida é validada para todos os estados de es, e o conectivo F é implementado utilizando a função orCheck que verifica se para os próximos estados, pelo menos um estado de es valida a fórmula. Para o conectivo temporal X o verificador analisa se para o próximo estado dentro de es, a propriedade é validada e já para o conectivo U é verificado

a validação da propriedade a esquerda do operador nos estados até que a propriedade a direita torna-se válida para os estados seguintes. Para o conectivo W fizemos uso da equivalência entre as fórmulas da LTL.

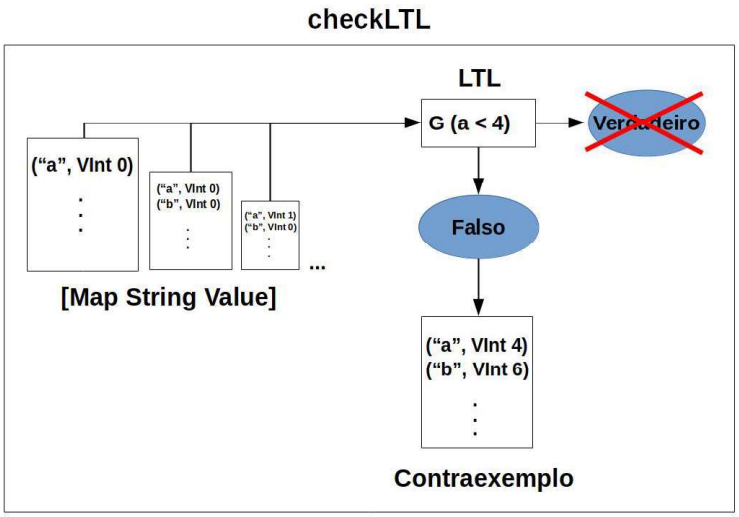
Figura 14 – Função checkLTL

```
checkLTL :: Interp a -> LTL -> [Env a] -> (Bool,Maybe (Env a))
checkLTL interp TT es        = (True,Nothing)
checkLTL interp FF es        = (False,Just $ head es)
checkLTL interp (Atom s)  es = let r = verifyAtom interp s es
                                in (r,mkMaybe r (head es))
checkLTL interp (l :&: r) es = (checkLTL interp l es) |&> (checkLTL interp r es)
checkLTL interp (l :|: r) es = (checkLTL interp l es) |+> (checkLTL interp r es)
checkLTL interp (Not l)   es = (|!>) (checkLTL interp l es)
checkLTL interp (X l) (_:xs) = checkLTL interp l xs
checkLTL interp (G l) es     = lstCheck $ map (checkLTLZ interp l . (:[])) es
checkLTL interp (F l) es     = orCheck  $ map (checkLTLZ interp l . (:[])) es
checkLTL interp (U l r) ss   = let ss' = dropWhile (fst.(checkLTL interp l . (:[]))) ss
                               in checkLTL interp r ss'
checkLTL interp (W l r) ss   = checkLTL interp ((U l r) :|: (G l)) ss
```

Fonte: Elaborada pelo autor.

Utilizando o mesmo exemplo de trecho de código, a verificação das propriedades LTL são realizadas como na figura abaixo:

Figura 15 – Exemplo de funcionamento do checkLTL



Fonte: Elaborada pelo autor.

Verificamos ao longo da lista de mapeamentos, se a variável "a"é sempre menor que 4, utilizando o conectivo G (*always*). Como no exemplo a variável em questão é incrementada enquanto seu valor for menor que 5, quando a mesma possuir o valor 4, seu estado não é validado para a propriedade e então, ele é retornado como contraexemplo.

4 Apresentação e análise dos resultados

Quando programamos para sistemas reativos é comum utilizarmos funções típicas como leitura e escrita de pinos, função de delay, configuração de pinos de entrada ou saída de dados, entre outras. De modo a representá-las em nossa EDSL imperativa, foram criadas declarações para elas no módulo AST (dentro do tipo Stmt), bem como funções da EDSL para cada uma e suas devidas execuções no módulo Interpretador.

Consideramos a existência de 32 pinos (1 - 32) que podem ser configurados tanto como entrada ou saída de dados. Tais pinos são representados como variáveis e, por padrão, são configurados como pinos de saída de dados, no entanto temos a opção de definir quais pinos serão configurados como de entrada de dados, e os que não são, continuam sendo de saída.

Para verificar a usabilidade da ferramenta desenvolvida, realizamos testes com escopos simples, próximos aos programadores de sistemas reativos, que serão apresentados nas seções seguintes.

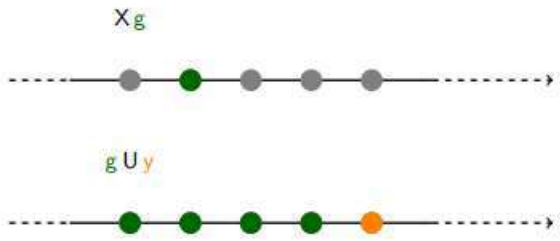
4.1 Semáforos

O comportamento de um semáforo pode ser modelado utilizando a lógica temporal linear conforme o conjunto de fórmulas abaixo:

$$
\begin{aligned}
\Gamma = \{ & \\
& G(\text{verde} \vee \text{amarelo} \vee \text{vermelho}), && (1)\\
& G(\text{verde} \Rightarrow (\ \neg\text{amarelo} \wedge \neg\text{vermelho})), && (2)\\
& G(\text{amarelo} \Rightarrow (\ \neg\text{verde} \wedge \neg\text{vermelho})), && (3)\\
& G(\text{vermelho} \Rightarrow (\ \neg\text{verde} \wedge \neg\text{amarelo})), && (4)\\
& G(\text{verde} \Rightarrow X\ \text{amarelo})), && (5)\\
& G(\text{amarelo} \Rightarrow X\ \text{vermelho})), && (6)\\
& G(\text{vermelho} \Rightarrow X\ \text{verde})) && (7)\\
\}&
\end{aligned}
$$

Este conjunto de fórmulas descrevem propriedades as quais um semáforo obrigatoriamente deve atender, como por exemplo sempre ter pelo menos uma luz acesa especificado por (1), ter somente uma luz acesa especificado por (2), (3) e (4), e definir uma sequência padrão das cores definida por (5), (6) e (7). Podemos também verificar outras propriedades utilizando outros conectivos temporais como na figura 16 que ilustra o funcionamento de dois conectivos temporais aplicados ao modelo de semáforo, o X (*next*) que especifica que o próximo estado é a cor verde, e o U (**until**) que especifica que a cor verde continua acesa nos próximos estados até o momento em que a amarela se acende.

Figura 16 – Conectivos temporais em um semáforo



Fonte: (RAMOS, 2011).

Um exemplo mais interessante para a verificação de propriedades da lógica temporal linear é a modelagem de dois semáforos sincronizados para controlar o fluxo de veículos em um cruzamento de pistas. O sincronismo entre os sinais significa que enquanto um estiver aberto, luz verde, ou em estado de atenção, luz amarela, o outro deve estar fechado e vice versa. Podemos construir este projeto utilizando o Arduíno, com o código de exemplo a seguir, cujo método loop é executado repetidamente enquanto a aplicação estiver funcionando.

Figura 17 – Algoritmo para o controle de dois semáforos

```
void setup(){
    // definindo os pinos digitais 5,6,7,8,9 e 10 como pinos de saida
    pinMode(5,OUTPUT);
    pinMode(6,OUTPUT);
    pinMode(7,OUTPUT);
    pinMode(8,OUTPUT);
    pinMode(9,OUTPUT);
    pinMode(10,OUTPUT);
}

void loop(){
    digitalWrite(5,LOW);    // apaga sinal vermelho (sinal 1)
    digitalWrite(7,HIGH);   // acende sinal verde (sinal 1)
    digitalWrite(8,HIGH);   // acende sinal vermelho (sinal 2)
    delay(4000);            // espera 4 segundos
    digitalWrite(7,LOW);    // apaga sinal verde (sinal 1)
    digitalWrite(6,HIGH);   // acende sinal amarelo (sinal 1)
    delay(2000);            // espera 2 segundos
    digitalWrite(6,LOW);    // apaga sinal amarelo (sinal 1)
    digitalWrite(5,HIGH);   // acende sinal vermelho (sinal 1)
    digitalWrite(8,LOW);    // apaga sinal vermelho (sinal 2)
    digitalWrite(10,HIGH);  // acende sinal verde (sinal 2)
    delay(4000);            // espera 4 segundos
    digitalWrite(10,LOW);   // apaga sinal verde (sinal 2)
    digitalWrite(9,HIGH);   // acende sinal amarelo (sinal 2)
    delay(2000);            // espera 2 segundos
    digitalWrite(9,LOW);    // apaga sinal amarelo (sinal 2)
}
```

Fonte: Elaborada pelo autor.

O semáforo 1 é representado pelos pinos 5, 6 e 7, e o semáforo 2 é representado pelos pinos 8, 9, e 10 das cores vermelho, amarelo e verde nesta ordem para ambos sinais de trânsito.

Na figura 18 temos o código da EDSL imperativa escrita com base no código fonte. Vale lembrar que os valores HIGH e LOW são voltagens de 5 e 0 respectivamente, e que não precisamos configurar os pinos como de saída de dados, uma vez que os mesmos já vêm configurados por padrão. Para representar o método loop, inserimos o seu conteúdo dentro de um laço de repetição.

Figura 18 – Algoritmo escrito com a EDSL

```
semaforos = do
              while (bool True)
              (do
                 writePin 5 (int 0)
                 writePin 7 (int 5)
                 writePin 8 (int 5)
                 delay(40)
                 writePin 7 (int 0)
                 writePin 6 (int 5)
                 delay(20)
                 writePin 6 (int 0)
                 writePin 5 (int 5)
                 writePin 8 (int 0)
                 writePin 10 (int 5)
                 delay(40)
                 writePin 10 (int 0)
                 writePin 9 (int 5)
                 delay(20)
                 writePin 9 (int 0)
                 writePin 8 (int 5))
```

Fonte: Elaborada pelo autor.

A AST é então montada e passada para o interpretador. Para que o interpretador não fique em um loop, uma vez que não conseguimos indicar que o sistema parou de funcionar, pegamos apenas uma sequência finita de repetições para então verificar algumas propriedades LTL dentro da execução traçada.

Podemos verificar propriedades que os sinais devem atender individualmente como dito inicialmente nesta seção, mas também outras propriedades, por exemplo, "uma vez vermelha, o sinal verde sempre torna-se aceso eventualmente após o sinal amarelo ter sido aceso por algum tempo"(BAIER; KATOEN, 2008). Expresso como:

$$\mathbf{G}(\text{vermelho} \rightarrow \mathbf{X}\ (\text{vermelho}\ \mathbf{U}\ (\text{amarelo} \wedge \mathbf{X}\ (\text{amarelo}\ \mathbf{U}\ \text{verde}))))$$

ou entao:

$$\mathbf{G}(\neg\text{vermelho} \vee \mathbf{X}\ (\text{vermelho}\ \mathbf{U}\ (\text{amarelo} \wedge \mathbf{X}\ (\text{amarelo}\ \mathbf{U}\ \text{verde}))))$$

Ademais, podemos verificar propriedades que devem ser validadas considerando o estado dos dois sinais, como os exemplos abaixo:

- os sinais de mesma cor verde ou amarelo nunca podem estar acesos juntos (e.g., G (pin7=5 →pin10=0))
- se a cor verde está acesa em um sinal, o outro deve obrigatoriamente estar vermelho (e.g, G (pin7=5 → pin8=5))
- se um sinal está na cor vermelha, o outro está com a cor verde ou amarela acesa (e.g., G (pin8=5 → (pin5=5 ∧ pin6=5)))

Para todas estas propriedades testadas, o verificador retornou verdadeiro para o algoritmo de entrada, porém tal resultado não significa que o programa está 100% correto, uma vez que o teste não cobre todas as possibilidades de execução.

Se realizarmos uma mudança no código de controle dos dois semáforos podemos notar a invalidação de alguma propriedade e verificar o contraexemplo devolvido. Consideramos a seguinte mudança:

Figura 19 – Alteração no código fonte dos semáforos

```
semaforos = do
            while (bool True)
            (do
               writePin 5 (int 0)
               writePin 7 (int 5)
               writePin 9 (int 5) -- alteracao do pino 8
               delay(40)          -- para o pino 9
               writePin 7 (int 0)
               writePin 6 (int 5)
               delay(20)
               writePin 6 (int 0)
               writePin 5 (int 5)
               writePin 8 (int 0)
               writePin 10 (int 5)
               delay(40)
               writePin 10 (int 0)
               writePin 9 (int 5)
               delay(20)
               writePin 9 (int 0)
               writePin 8 (int 5))
```

Fonte: Elaborada pelo autor.

Com essa alteração, induzimos um erro na sincronização dos semáforos e, por isso, algumas propriedades verificadas anteriormente, já não serão mais validadas. Uma delas é a fórmula LTL a seguir:

$$G\ (pin6{=}5 \rightarrow pin9{=}0)$$

a qual especifica que o sinal amarelo não pode estar aceso nos dois semáforos. Logo, o verificador retornou falso para esta propriedade e o contraexemplo foi gerado.

Figura 20 – Contraexemplo

```
(False,Just (fromList [("01p",VInt 0),("02p",VInt 0),
                       ("03p",VInt 0),("04p",VInt 0),
                       ("05p",VInt 0),("06p",VInt 5),
                       ("07p",VInt 0),("08p",VInt 0),
                       ("09p",VInt 5),("10p",VInt 0),
                       ("11p",VInt 0),("12p",VInt 0),
                       ("13p",VInt 0),("14p",VInt 0),
                       ("15p",VInt 0),("16p",VInt 0),
                       ("17p",VInt 0),("18p",VInt 0),
                       ("19p",VInt 0),("20p",VInt 0),
                       ("21p",VInt 0),("22p",VInt 0),
                       ("23p",VInt 0),("24p",VInt 0),
                       ("25p",VInt 0),("26p",VInt 0),
                       ("27p",VInt 0),("28p",VInt 0),
                       ("29p",VInt 0),("30p",VInt 0),
                       ("31p",VInt 0),("32p",VInt 0)]))
```

Fonte: Elaborada pelo autor.

Podemos observar que os valores do pino 6 e 9 que representam o sinal amarelo estão ambos acesos (VInt 5) em um estado do programa e isto invalida a propriedade e, portanto, o algoritmo apresenta um erro quanto a modelagem de sincronismo.

4.2 Sensor de temperatura

Quando a aplicação possui um código com leitura de valores de pinos vindos de um sensor de temperatura, por exemplo, usamos a função readPin da EDSL imperativa que gera um valor pseudo aleatório entre (0,255), assumindo uma resolução de 8 bits, por meio da biblioteca System.Random nativa do Haskell, e retorna este valor como resultado da função.

Consideramos um código de exemplo abaixo, no qual é definido o pino 1 como de entrada de dados, cujo valor de leitura vem de um sensor de temperatura, e este valor é armazenado em uma variável temperatura.

Podemos então querer verificar se em algum estado futuro, o valor do pino 5 de saída que recebe o valor lido do sensor estará sempre dentro de um intervalo de temperaturas

Figura 21 – Código de leitura para um sensor de temperatura

```
temperatura = do
              declare "temp" TyInt (int 0)
              inputPin 1
              while (bool True)
                    (do
                       readPin 1 "temp"
                       writePin 5 (var "temp"))
```

Fonte: Elaborada pelo autor.

predeterminado, conforme a fórmula LTL abaixo:

$$G\ (\text{"pin5"} > \text{min} \wedge \text{"pin5"} < \text{max})$$

ou então poderíamos configurar que após um determinado valor de temperatura, o próximo valor lido é esperado que seja mais baixo que um limiar, por exemplo.

$$X\ (\text{"pin5"} < \text{limiar})$$

Tais propriedades podem ser definidas pelo programador do sistema reativo de acordo com o contexto o qual a aplicação estará inserida.

5 Conclusão

O desenvolvimento do presente estudo possibilitou a criação de uma ferramenta de testes de software para sistemas reativos utilizando propriedades especificadas da lógica temporal linear. O suporte para testes adequado para o desenvolvimento de sistemas reativos pode reduzir drasticamente o custo de produção de tais sistemas e, consequentemente, aumentar sua qualidade. É sabido que grande parte do esforço do desenvolvimento de um software está em etapas de validação e teste e elas costumam representar cerca 50% do custo final do sistema. Nesse sentido, o presente trabalho trás um sistema de validação alternativo para desenvolvedores de sistemas reativos.

Em ambos os testes realizados dos códigos de exemplo, conseguimos verificar as propriedades temporais e gerar um contraexemplo simples que mostram ao desenvolvedor o primeiro estado o qual pelo menos uma propriedade não foi validada.

Dada à importância da realização de testes e por hoje em dia, a construção e execução deles serem uma tarefa dispendiosa, é viável o aprimoramento da ferramenta criada por este trabalho e dar continuidade ao seu desenvolvimento, como dar suporte, na EDSL imperativa criada, para novas estruturas, por exemplo funções, registros, ponteiros e outras comuns em linguagens imperativas, aperfeiçoar o contraexemplo devolvido para o desenvolvedor, mostrando não apenas o estado com a qual uma propriedade não foi validada, e sim todo o traço do programa que levou àquele estado, e elaborar um método mais eficiente de geração de valores aleatórios para leituras de pinos de entrada de dados.

Referências

BAIER, C.; KATOEN, J.-P. *Principles of model checking*. [S.l.]: MIT press, 2008. Citado 5 vezes nas páginas 16, 19, 21, 22 e 37.

BENTLEY, J. Programming pearls: Little languages. *Commun. ACM*, ACM, New York, NY, USA, v. 29, n. 8, p. 711–721, ago. 1986. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/6424.315691>. Citado na página 23.

BÖRGER, E.; STÄRK, R. *Abstract state machines: a method for high-level system design and analysis*. [S.l.]: Springer Science & Business Media, 2012. Citado na página 16.

CLAESSEN, K.; HUGHES, J. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, ACM, v. 46, n. 4, p. 53–64, 2011. Citado 2 vezes nas páginas 15 e 24.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 8, n. 2, p. 244–263, 1986. Citado na página 21.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. *Model checking*. [S.l.]: MIT press, 1999. Citado 2 vezes nas páginas 16 e 21.

DEURSEN, A. van; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 6, p. 26–36, jun. 2000. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/352029.352035>. Citado na página 23.

FERREIRA, N. F. G. *Verificação formal de sistemas modelados em estados finitos*. Tese (Doutorado) — Universidade de São Paulo, 2005. Citado na página 16.

HOLZMANN, G. *Spin model checker, the: primer and reference manual*. [S.l.]: Addison-Wesley Professional, 2003. Citado na página 16.

HUGHES, J. Why functional programming matters. *The computer journal*, Oxford University Press, v. 32, n. 2, p. 98–107, 1989. Citado na página 25.

JONES, J. Abstract syntax tree implementation idioms. In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. [S.l.: s.n.], 2003. p. 1–10. Citado na página 26.

JONES, S. P. *Haskell 98 language and libraries: the revised report*. [S.l.]: Cambridge University Press, 2003. Citado na página 15.

LIPOVACA, M. *Learn you a haskell for great good!: a beginner's guide*. [S.l.]: no starch press, 2011. Citado 2 vezes nas páginas 27 e 28.

MARKETS; MARKETS. Embedded systems market by hardware (mpu, mcu, application specific ic / application specific standard product, dsp, fpga, and memory), software (middleware and operating system), application, and geography - global forecast to 2023. 2017. Citado na página 17.

MYERS, G. J. *The art of software testing.* [S.l.]: John Wiley & Sons, 2006. Citado na página 15.

PNUELI, A. The temporal logic of programs. In: IEEE. *Foundations of Computer Science, 1977., 18th Annual Symposium on.* [S.l.], 1977. p. 46–57. Citado na página 16.

RAMOS, J. Lógica temporal e aplicaçoes. 2011. Citado na página 36.

SOMMERVILLE, I. *Software Engineering (5th Ed.).* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-42765-6. Citado na página 17.

TERMO DE RESPONSABILIDADE

Eu, **Matheus Takeshi Yamakawa Ikeda** declaro que o texto do trabalho de conclusão de curso intitulado "***Teste de software baseado em lógica temporal linear em sistemas reativos***" é de minha inteira responsabilidade e que não há utilização de texto, material fotográfico, código fonte de programa ou qualquer outro material pertencente a terceiros sem as devidas referências ou consentimento dos respectivos autores.

João Monlevade, 11 de julho de 2018

Matheus Ikeda

Matheus Takeshi Yamakawa Ikeda

ANEXO IX – DECLARAÇÃO DE CONFORMIDADE

Certifico que o(a) aluno(a) **Matheus Takeshi Yamakawa Ikeda**, autor do trabalho de conclusão de curso intitulado **"Teste de software baseado em lógica temporal linear em sistemas reativos"** efetuou as correções sugeridas pela banca examinadora e que estou de acordo com a versão final do trabalho.

João Monlevade, 27 de junho de 2019.

Eltan M. Cardoso
Professor (a) Orientador (a)