UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Verificação de Programas Multi-Tarefas Baseado no Framework Multiplataforma QT

Adriana Silva de Souza

MANAUS
2019

UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Adriana Silva de Souza

Verificação de Programas Multi-Tarefas Baseado no Framework Multiplataforma QT

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica, área de concentração Controle e Automação de Sistemas.

Orientador: Prof. Dr. Lucas C. Cordeiro

MANAUS
2019

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

S729v

Souza, Adriana Silva de

Verificação de Programas Multi-Tarefas Baseado no Framework Multiplataforma QT / Adriana Silva de Souza. 108

CVIII f.: il. color; 31 cm.

Orientador: Lucas C. Cordeiro

Dissertação (Mestrado em Engenharia Elétrica) - Universidade Federal do Amazonas.

1. Modelo Operacional. 2. Verificação Formal. 3. Multi-Tarefas. 4. Solucionadores. I. Cordeiro, Lucas C. II. Universidade Federal do Amazonas III. Título

ADRIANA SILVA DE SOUZA

VERIFICAÇÃO DE PROGRAMAS MULTI-TAREFAS BASEADO NO FRAMEWORK MULTIPLATAFORMA QT

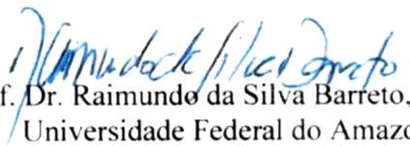
Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica na área de concentração Controle e Automação de Sistemas.

Aprovado em 03 de setembro de 2019.

BANCA EXAMINADORA

Prof. Dr. Lucas Carvalho Cordeiro, Presidente
Universidade Federal do Amazonas

Prof. Dr. Eddie Batista de Lima Filho, Membro
R&D/TPV do Brasil



Prof. Dr. Raimundo da Silva Barreto, Membro
Universidade Federal do Amazonas

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me mantido com saúde e por todas as conquistas que me trouxeram até aqui.

Agradeço a minha mãe Raimunda Nonata pelo apoio, amor, suporte, incentivo e principalmente por nunca deixar de acreditar em mim. Obrigada por me motivar e incentivar a ir além do que eu imaginava conseguir chegar.

Agradeço aos meus irmão a quem tanto amo e que me enchem de orgulho, aos meus tios, primos e amigos, pessoas maravilhosas que não deixaram de acreditar e me incentivar, mesmo nos momentos mais difíceis.

Aos meus colegas de laboratório e sala de aula, os quais tive o prazer de partilhar as dúvidas, as preocupações e alegrias que tal experiência me proporcionou, os quais hoje tenho prazer de chamar de amigos. Compartilhar de tal experiência com vocês foi algo maravilhoso, obrigada.

Deixo o meu agradecimento especial ao professor Lucas Cordeiro, o qual tive o prazer de ter como orientador. Obrigada pelo suporte, direcionamento, encorajamento nos momentos difíceis e por nunca deixar de me incentivar, o que foi essencial para o desenvolvimento desta pesquisa. Deixo aqui o meu muito obrigada, sem sua ajuda essa pesquisa não seria possível.

À Fundação de Amparo à Pesquisa do Estado do Amazonas (FAPEAM) pelo apoio financeiro.

Esta pesquisa, conforme previsto no Art. 48 do decreto nº 6.008/2006, foi financiada pela Samsung Electrônica da Amazônia Ltda, nos termos da Lei Federal nº 8.387/1991, através de convênio nº 004, firmado com o CETELI/ UFAM.

Resumo

Com o avanço da tecnologia, progressos têm acontecido no que diz respeito ao desenvolvimento de hardware e software. O resultado de tais avanços é decorrente da evolução das aplicações que, atualmente, requerem mais capacidade de processamento e armazenamento de dados. A partir desta realidade, a indústria investe fortemente em processos de verificação rápidos e automáticos. Com o intuito de obter a diminuição das taxas de erros presentes nos sistemas produzidos, durante o processo de desenvolvimento é de fundamental importância utilizar técnicas adequadas para a implementação do paralelismo e da sincronização dos dados, para que possa ser extraído o desempenho máximo do sistema produzido. Tendo em vista que o tempo do processo de desenvolvimento está cada vez menor e a robustez dos sistemas tende a aumentar exponencialmente com o tempo, muitos *frameworks* têm sido utilizados no processo de desenvolvimento das empresas de sistemas embarcados, com o intuito de acelerar os processos de desenvolvimento. Um *framework*que tem se destacando é o de desenvolvimento multi-plataforma chamado Qt, que conta com um amplo conjunto de bibliotecas de fácil implementação, manutenção e multi-plataforma, para sistemas multi-tarefas. Sendo assim, o presente trabalho propõe um conjunto de bibliotecas simplificadas chamadas de Modelo Operacional Simplificado Multi-Tarefas Qt, similares à biblioteca padrão para a criação e manipulação de *threads* dentro do *framework* Qt. Estas bibliotecas foram integradas inicialmente ao verificador de *software Efficient SMT-Based Bounded Model Checking (ESBMC)* de forma que, através do modelo operacional proposto, o verificador ESBMC possa realizar a verificação de aplicações que utilizam o *framework Qt* para implementar programas concorrentes. A inclusão do modelo operacional nos processos de verificação do ESBMC mostrou-se uma excelente abordagem para a verificação de programas multi-tarefas Qt, tendo uma taxa de verificações bem sucedidas de 90%. O modelo operacional proposto mostrou sua versatilidade ao ser incluído nos processos de verificações de outros dois verificadores de modelos chamados DIVINE e LLBMC. O verificador de modelos DIVINE obteve uma taxa de verificações bem sucedidas de 87%, já o verificador de modelos LLBMC obteve uma taxa de 85%. Com isso, podemos validar a viabilidade da abordagem que propomos para verificar programas multi-tarefas Qt. Por fim, a metodologia que propomos encontra-se em estado da arte, sendo a primeira a viabilizar a verificação de estruturas multi-tarefas Qt, podendo ser expandido, para outras estruturas que ainda não foram cobertas.
Palavras-chave: Modelo Operacional, *Framework* Qt, Verificação Formal, *Bounded Model Checking*.

Abstract

With the advancement of technology, progress has been made with respect to the development of hardware and software. The result of such advances is due to the evolution of applications, which currently require more data processing and storage capacity. From this reality, the industry invests heavily in automatic and fast verification processes, in order to obtain the reduction of error rates present in the developed systems, which thus becomes necessary, during the process of development of these systems, the use of such techniques suitable for the implementation of parallelism and data synchronization, such as data race, so that it can be extracted, the maximum performance of the system being produced. With this in mind, the development process time and the robustness of these systems, are becoming smaller, with that the use of framework in the development process has been very used. A framework that is in evidence now is the Qt multi-platform development framework, which has a large set of libraries for easy deployment, maintenance and portability for multi-thread systems. In this dissertation, we propose a set of simplified Qt multi-threaded libraries, similar to the standard library for the creation and manipulation of threads within the Qt framework. These models are integrated with the Efficient SMT-Based Bounded Model Checker (ESBMC), so that through the operational models, it is possible to analyze real applications that use the Qt framework, which implements concurrent programming. Through the simplified Qt multi-threaded Operational Model, we can perform an evaluation of the methodology proposed with other model verifiers, with the purpose of evaluating the feasibility of the approach we propose. Thus, the methodology we propose here is in the state-of-the-art, being the first one to make feasible the verification of multi-thread structures of the Qt framework, and can be expanded, to other structures that have not yet been covered.

Keywords: Operational Model, Qt Framework, Bounded Model Checking, Formal Verification.

Lista de ilustrações

Figura 1 – Árvore de busca resultante da enumeração de todos os possíveis estados das variáveis A, B e C . (PIPATSRISAWAT; DARWICHE, 2009) 27
Figura 2 – Arquitetura do LLBMC (GADELHA et al., 2013). 29
Figura 3 – Exemplo de código para a criação de threads no Qt. 33
Figura 4 – Funcionamento da thread no framework Qt (COMPANY,) 34
Figura 5 – Grafo de Fluxo de Controle Direcionado 39
Figura 6 – Exemplo de três *threads* executando simultaneamente o método *run()*. . . . 54
Figura 7 – Grafo de Fluxo de controle do programa em formato GOTO de duas threads 55
Figura 8 – Descrição do funcionamento do Modelo Operacional Multi-Tarefas Qt. . . . 57
Figura 9 – Processo de desenvolvimento do Modelo Operacional Multi-Tarefas QT. . . 60
Figura 10 – Representação do Escopo das Fases de desenvolvimento do Modelo Operacional *Multi-Tarefas Qt*. 63
Figura 11 – Exemplo de validação da criação do objeto de classe, com o auxílio da inclusão de assertivas. 64
Figura 12 – Struct com variáveis compartilhadas *CriticalSection*, usadas no gerenciamento dos recursos compartilhados. 66
Figura 13 – Representação do Funcionamento do Espaço de Endereçamento no Modelo Operacional *Multi-Tarefas Qt*. 67
Figura 14 – Principais estados que a *thread* pode assumir em tempo de execução. 73
Figura 15 – Escopo principal da Seção Crítica. 76
Figura 16 – Escopo da Seção Crítica. 77
Figura 17 – Assinatura das funções da Área de Exclusão Mútua 78
Figura 18 – Assinatura do método *start()*, no *framework* Qt. 78
Figura 19 – Assinatura do enumerador *Priority*, no *framework* Qt. 79
Figura 20 – Representação do método *start()*, no modelo operacional proposto. 79
Figura 21 – Exemplo da implementação do método *start* com atribuição de prioridade. . 80
Figura 22 – Exemplo de ocorrência de condição de corrida com violação de atomicidade. 81
Figura 23 – Exemplo de modelagem de atomicidade em declarações visíveis. 82
Figura 24 – Resultados da análise dos Tempos de Verificação do ESBMC utilizando os solucionadores Boolector, Z3 e Yices 2 executados em conjunto com o verificador de modelos ESBMC. 89
Figura 25 – Resultados da análise da Taxa de Cobertura das verificações do ESBMC utilizando os solucionadores Boolector, Z3 e Yices 2 executados em conjunto com o verificador de modelos ESBMC. 90

Figura 26 – Comparação entre os tempos de verificação das suítes de testes, resultantes da verificação do modelo operacional simplificado com os verificadores de modelos ESBMC++, LLBMC e DIVINE. 93
Figura 27 – Comparação entre as taxas de verificação das suítes de testes, resultantes da verificação do modelo operacional simplificado com os verificadores de modelos ESBMC++, LLBMC e DIVINE. 94
Figura 28 – Escopo das variáveis globais 97
Figura 29 – Escopo do código para a classe *Producer*. 98
Figura 30 – Escopo do código para a classe Consumer 99
Figura 31 – Representação do main() 100
Figura 32 – Representação de um método reimplementado de QObject 100
Figura 33 – Diagrama de atividades desenvolvidas no desenvolvimento do modelo operacional simplificado multi-tarefas QT. 102
Figura 34 – Diagrama de atividades desenvolvidas para a validação da conformidade do Modelo Operacional Simplificado Multi-Tarefas QT. 103

Lista de tabelas

Tabela 1 – Tabela Verdade 24
Tabela 2 – Tabela de resultados- Verificadores de modelos 49
Tabela 3 – Na tabela a baixo e possível observar os tipos de prioridades que uma thread pode assumir no momento de sua execução. 67
Tabela 4 – Tabela de resultados- Verificadores de modelos 92

Lista de Abreviaturas e Siglas

UFAM	Universidade Federal do Amazonas
FIFO	First in, first out
SJF	Shortest Job First
SRT	Shortest Remaining Time
RR	Round-Robin
LP	Lógica Proposicional
SAT	SATisfatibilidade Booleana
SMT	Satisfiability Modulo Theories
VC	Verification Condition
CBMC	C Bounded Model Checker
ESBMC	Efficient SMT-Based Context-Bounded Model Checker
STL	Standard Template Library
LP	Lógica Proposicional
CNF	Conjective Normal Form
AST	Intermediate REPresentation
IREP	Intermediate REPresentation
VCG	Verification Ccondition Generator
SSA	Single Static Assignment
CFG	Control-Flow Graph
LLBMC	Low Level Bounded Model Checker
LLVM IR	Representação Intermediaria LLVM
LLVM ILR	Representação Intermediária Lógica LLVM
WCET	worst-case execution time

Sumário

1	**INTRODUÇÃO**	**14**
1.1	**Área de Aplicação e Descrição dos Problemas**	**16**
1.2	**Objetivos**	**18**
1.3	**Descrição da Solução**	**19**
1.4	**Contribuições**	**20**
1.5	**Organização da Dissertação**	**21**
2	**FUNDAMENTAÇÃO TEÓRICA**	**22**
2.1	**Fundamentos Lógicos**	**22**
2.1.1	Lógica Proposicional	22
2.1.1.1	Davis-Putnam-Logemann-Loveland(DPLL)	25
2.1.2	Bounded Model Checking (BMC)	26
2.1.3	Efficient SMT-based Context-Bounded Model Checker	28
2.1.4	Low-Level Bounded Model Checker(LLBMC)	28
2.1.5	Técnicas de Verificação de Modelos	30
2.2	**Tecnologias Multi-Tarefas no Qt**	**32**
2.2.1	Verificação de Sistemas Multi-Tarefas	34
2.2.2	Modelagem de Sistemas Multi-Tarefas em Qt	36
2.2.3	Concorrência e Intercalação	36
2.3	**Resumo**	**39**
3	**TRABALHOS RELACIONADOS**	**41**
3.1	**Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking**	**42**
3.2	**Verifying Multi-threaded C Programs with SPIN**	**43**
3.3	**Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization**	**44**
3.4	**Efficient Modeling of Concurrent Systems in BMC**	**45**
3.5	**Verifying Atomicity Specifications for Concurrent Object-Oriented Software Using Model-Checking**	**46**
3.6	**Checking Concise Specifications For Multithreaded Software**	**47**
3.7	**Resumo**	**48**
4	**VERIFICAÇÃO DE PROGRAMAS MULTI-TAREFAS QT**	**50**
4.1	**Introdução**	**51**
4.2	**Conceitos Preliminares**	**52**

4.2.1 Programas Qt Multi-Tarefas 53
4.3 Verificação de Modelo Limitado por Contexto de Software Qt Multi-tarefas 56
4.3.1 Especificação de Propriedade 61
4.3.2 Explorando a Árvore de Acessibilidade 64
4.3.3 Fluxo de Controle 68
4.3.4 Threads Lançadas em Tempo de Execução e *Link* de Ponteiros 69
4.3.5 Detectando Corridas de Dados e Proteção de memória 70
4.4 Concorrência e Acesso à Memória Compartilhada 72
4.4.1 Memória escalonável 72
4.5 Modelando Primitivas de Sincronização no *QThread* 74
4.5.1 Modelando Operações de Bloqueio de *QThread* 75
4.5.2 Agendamento de Prioridades no Qt 76
4.5.3 Modelando Operações de Bloqueio e Exclusão Mútua 78
4.5.4 Modelando as Condições de Corrida é Sinalizadores 79
4.5.5 Resumo 82

5 AVALIAÇÃO EXPERIMENTAL 84
5.1 Configuração dos Experimentos 84
5.2 Uma Comparação Entre os Solucionadores SMT (Z3, Boolector e Yices 2) 87
5.3 Avaliação dos Resultados Para o Modelo Operacional Simplificado Multi-Tarefas 89
5.4 Avaliação Experimental - Análise do Modelo Operacional 95
5.4.1 Escolhas Determinísticas e Contra-Exemplos 95
5.4.2 Condições de Verificação 96
5.4.3 Pré-condições - Verificação dos Resultados Para os *benchmarks* 96
5.5 Avaliação de Conformidade do Operacional Simplificado Multi-Tarefas Qt/C++ 101

6 CONCLUSÕES 104
6.1 Trabalhos Futuros 105
6.2 Agradecimentos 106

Referências 107

1 Introdução

O mercado de sistemas embarcados está em ampla expansão. A todo o momento, novos sistemas são desenvolvidos, cada vez mais complexos, robustos, com processadores de vários núcleos e memórias escalonáveis. Com isso, para evitar o desperdício de recursos do *hardware* e de *software*, torna-se necessário, durante o processo de desenvolvimento desses sistemas, a utilização de técnicas adequadas para a implementação do paralelismo e da sincronização dos dados (CORDEIRO, 2010; CORDEIRO; FISCHER, 2011; CORDEIRO, 2011), tais como a programação concorrente, para que possa ser extraído o máximo desempenho do sistema que está sendo produzido.

Com isso, visando à grande demanda dos sistemas que precisam ser desenvolvidos de forma rápida, barata e com um alto grau de confiabilidade, muitos *frameworks* têm sido utilizados com o objetivo de acelerar o desenvolvimento dos sistemas que são produzidos. É exatamente nesse contexto que o *framework Qt* (The Qt Company Ltd., 2015) vem despontando como um excelente conjunto de classes reutilizáveis, podendo ser utilizado no processo de desenvolvimento de sistemas e bibliotecas multi-plataformas. Em especial, o *framework Qt* pode ser executado em diversos ambientes diferentes sem a necessidade da alteração do código-fonte, o que torna possível o desenvolvimento de aplicações multi-plataformas de forma rápida e fácil, o que o torna uma excelente ferramenta no processo de desenvolvimento dos sistemas embarcados.

Partindo da demanda crescente de processos de produção mais rápidos, baratos e de alta qualidade, a indústria tem investido fortemente em mecanismos de desenvolvimento, que podem tornar o processo de projetar, desenvolver, manter e testar os sistemas de forma rápida e o mais eficiente possível (SOUSA; CORDEIRO; FILHO, 2015; GARCIA et al., 2016; MONTEIRO et al., 2017; MONTEIRO et al., 2018).

Embora o *Qt* forneça mecanismos de baixo e de alto nível para a implementação de programas concorrentes (BEYER, 2015), seus módulos são de difícil depuração, o que dificulta os processos de testes. Isso ocorre em virtude de seu algoritmo ser não determinístico (sinalizadores e o conjunto de bibliotecas padrões) e a sua estrutura hierárquica ser grande e complexa, o que, durante o processo de verificação pode ocasionar um potencial aumento do espaço de estados, podendo gerar falhas de segmentação durante o processo de verificação dessas estruturas.

De modo geral, a programação concorrente é de difícil implementação, uma vez que a concorrência introduz uma série de novas classes de defeitos que pode acarretar em problemas graves durante a execução dos sistemas produzidos. Dentre os defeitos estão: condições de corrida, *deadlock*, *starvation*, *interleaving*, acesso indevido da memória, sincronização de dados, compartilhamento de dados, falhas de escalonamento, erros durante as trocas de contextos, entre outros (PEREIRA et al., 2016; PEREIRA et al., 2017; MONTEIRO et al., 2018). Com isso, a detecção e a localização de falhas nos sistemas produzidos são fundamentais durante o processo de desenvolvimento dos sistemas embarcados (ALVES, 2018; ROCHA et al., 2012; ALVES;

CORDEIRO; FILHO, 2015; ALVES; CORDEIRO; FILHO, 2017). É nesse cenário que na indústria tem surgido um crescente aumento do uso de técnicas de verificação para garantir a confiabilidade dos sistemas, com o intuito de detectar e resolver problemas no decorrer do processo de desenvolvimento (GADELHA et al., 2018).

Uma das formas mais eficazes e de menor custo para garantir a confiabilidade dos sistemas encontra-se na utilização das técnicas de verificação formal de sistemas baseados em Verificação de Modelos (do inglês, *Model Checking*(MC)) (MYERS; SANDLER; BADGETT, 2011).

A MC é uma técnica muito utilizada no processo de verificação automática de sistemas. Através da sua utilização é possível realizar a modelagem de sistemas e especificar as propriedades a serem validadas mediante a utilização de provas de teoremas (usualmente modeladas a partir de fórmulas lógicas).

As propriedades de sistema usualmente são validadas por intermédio da utilização de máquinas de estados, mediante a utilização de teorias do módulo da satisfatibilidade (do inglês,Satisfiability Modulo Theories(SMT)) .

Embora a verificação baseadas em MC seja muito eficaz, nem todos os sistemas podem ser verificados de forma automática, devido à indisponibilidade de verificadores capazes de reconhecer a linguagem que será alvo da verificação (ALVES, 2018; JANUÁRIO et al., 2014; MONTEIRO et al., 2017).

Considerando a complexidade da semântica do *Qt*, uma boa estratégia para realizar a verificação de suas estruturas é com a utilização de modelos operacionais, que nada mais são do que estruturas mais simples de uma determinada linguagem de programação, permitindo, assim, que por intermédio dessas estruturas possa ser possível reproduzir o comportamento da linguagem que será alvo da verificação, de forma que as propriedades da linguagem possam ser re-implementadas e verificadas de forma precisa e livres de ambiguidades, e, por fim, mediante ao uso de métodos de prova de teoremas, a fim de que todas as propriedades do sistema possam ser validadas (SOUZA; CORDEIRO; JANUARIO, 2018).

A técnica Verificação de Modelos Limitados(do inglês *Bounded Model Checking* (BMC)) (GANAI; GUPTA, 2008) é uma técnica de MC que tem sido amplamente utilizada na detecção de falhas em programas concorrentes, uma vez que são essencialmente não determinísticos. A partir de sua utilização é possível restringir o comprimento da árvore que será gerada durante o processo de verificação, evitando a ocorrência de explosões de espaço de estados durante o processo de análise de programas concorrentes.

Durante o processo de verificação de programas concorrentes, as entradas do sistema podem gerar saídas diferentes, tendo em vista ainda a possibilidade de gerar condições de corrida, uma vez que dada uma propriedade do sistema, é possível verificar a sua corretude para qualquer estado que o programa possa vir a assumir durante o processo de verificação, o que nos permite garantir se uma determinada prioridade está sendo atendida corretamente.

Com isso, considerando a complexidade da linguagem de programação *Qt*, propomos um Modelo Operacional Simplificado Multi-Tarefas Qt, contendo apenas as principais estruturas

de dados responsáveis pela implementação do paralelismos e da concorrência de dados, tais estruturas são implementadas com o auxílio dos módulos presentes na biblioteca *Qthreads* (The Qt Company Ltd. Documentation, 2015).

Mediante o modelo desenvolvido, foi possível integrá-lo no processo de compilação do verificador de *software* ESBMC (CORDEIRO et al., 2012; MORSE et al., 2013; MORSE et al., 2014), por meio de tal integração, o ESBMC foi capaz de reconhecer e validar de forma automática todas as propriedades que foram desenvolvidas no modelo, de tal forma, foi possível garantir a corretude dos sistemas a serem avaliados. Tais propriedades puderam ser validadas mediante a inclusão de pré e pós-condições que foram adicionadas ao longo do Modelo Operacional Simplificado Multi-Tarefas Qt.

No presente trabalho, aplicamos a técnica BMC em um contexto de linguagens imperativas, onde a ideia principal baseia-se na inclusão de *asserts* ao longo do Modelo Operacional, tendo como objetivo principal a validação e a checagem de propriedades inerentes da linguagem que, obrigatoriamente, devem ser satisfeitas em todas as execuções do modelo.

A versatilidade da metodologia que propomos para o desenvolvimento do Modelo Operacional Simplificado Multi-Tarefas Qt pode ser comprovada com o auxílio da inclusão do modelo em outros verificadores baseados em BMC que, assim como o ESBMC, suportam a verificação de programas multi-tarefas escritos em C/C++. Sendo assim, pudemos comprovar que o modelo que propomos é portável a verificadores de modelos com suporte à verificação de programas concorrentes desenvolvidos em C/C++.

1.1 Área de Aplicação e Descrição dos Problemas

A principal área de aplicação que a pesquisa apresentada aborda é a verificação formal de software eficiente e minimamente dispendiosa de programas *Qt/C++*, que implementem a concorrência e o paralelismo em programas multi-tarefas Qt.

Tal verificação pode tornar-se possível, através do desenvolvimento de um modelo operacional simplificado contendo apenas as principais estruturas de dados que permitem o desenvolvimento de programas multi-tarefas Qt, de tal forma que o modelo operacional possa ser adicionado de maneira implícita aos processos de compilação de verificadores de modelos baseados em BMC, capazes de prestar suporte a verificação de programas concorrentes desenvolvidos em *C/C++*.

Com isso, é possível adicionar aos verificadores novas funcionalidades, uma vez que, outrora, os mesmos não eram capazes de reconhecer e realizar a verificação de tais estruturas de dados presentes na linguagem de programação *Qt*.

Este trabalho aborda dois problemas da área de verificação de modelos: (1) a verificação de modelos de programas Qt/C++ e (2) verificação de programas multi-tarefas Qt.

O primeiro problema ocorre porque a verificação de modelos de programas escritos na linguagem Qt é uma tarefa difícil e complexa, uma vez que a linguagem de programação Qt

é composta por componentes de vários tamanhos, variando de funções e classes individuais, passando por unidades e bibliotecas, até *frameworks* e aplicativos completos (BARNAT et al., 2013). Neste contexto, o *framework Qt* apresentou uma série de problemas relacionados à complexidade de suas estruturas de dados, o que precisou ser considerado e tratado durante a construção do Modelo Operacional Simplificado Multi-Tarefas Qt, entre eles estão:

(1) Aumento da precisão da verificação, pensando no custo computacional com o intuito de evitar possíveis explosões do espaço de estados que podem ocorrer no momento da verificação;

(2) Houve a necessidade do processamento de grandes quantidades de instruções, levando em consideração a complexidade do *framework* alvo da verificação;

(3) O *framework* Qt conta com um conjunto amplo de bibliotecas, conexões internas (SOUSA, 2013) e principalmente interações implícitas (BARNAT et al., 2013), como as que serão o alvo das verificações, uma vez que as *threads* são muito difíceis de controlar mesmo em um ambiente de teste (BARNAT et al., 2013);

O segundo problema ocorre porque erros de concorrência são difíceis de detectar, pois ocorrem geralmente em tempo de execução. Uma alternativa utilizada para atacar essa problemática, foi identificar problemas de concorrências conhecidos na literatura (i.e., condições de corrida, deadloocks, falhas de segmentação, dentre outros) e aplicar as técnicas de verificação de software conhecidas para realizar as análises dos algoritmos e expor as falhas que podem ocorrer durante o processo de execução de programas multi-tarefas Qt/C++. Durante o processo de verificação de programas concorrentes Qt, os principais pontos que precisaram ser considerados e tratados durante a verificação de programas multi-Tarefas Qt foram:

(1) Erros de concorrência geralmente ocorrem sob intercalações de threads específicas;

(2) O número de intercalações cresce exponencialmente com o número de threads e instruções do programa;

(3) Mudanças de contexto entre threads aumentam o número de execuções possíveis;

Como alternativa para resolver o problema de possíveis explosões de espaços de estados e os possíveis indeterminismos que podem ocorrer durante a verificação de programas concorrentes, a presente pesquisa propõe uma abordagem de isolar as funções e os métodos que serão alvos das verificações de acordo com o nível de importância (Ex:. funções de criação, funções manipulações, funções destrutoras) que cada estrutura desempenha no funcionamento de programas multi-tarefas Qt, com isso, foi possível diminuir o custo e a complexidade das verificações, focando apenas em funções e métodos específicos.

Partindo daí, o modelo desenvolvido mostrou-se uma ferramenta robusta, uma vez que pode ser adicionada a diferentes verificadores BMC, tornando, assim, possível a verificação do modelo

proposto, com o auxílio de provas de teoremas automatizados e verificações decorrentes da utilização de modelos simbólicos, de acordo com a técnica aplicada em cada um dos solucionadores SMT e verificadores BMC que foram utilizados.

1.2 Objetivos

O principal objetivo desta dissertação é propor uma metodologia para verificar programas concorrentes Qt. Através da metodologia proposta, desenvolver um Modelo Operacional Simplificado Multi-Tarefas Qt que possa ser incorporado aos processos de identificações e verificações de verificadores de modelos baseados em BMC, para que os mesmos tornem-se capazes de reconhecer e verificar de forma automática estruturas de dados multi-tarefas Qt.

Os objetivos específicos são:

- Propor uma metodologia para verificação de programas concorrentes baseada na técnica MC, para que seja possível verificar a corretude de programas multi-tarefas Qt partir da utilização de verificadores de modelos BMC;

- Implementar um modelo operacional (e.g, verificar as heranças associadas aos métodos abordados no modelo, verificar e documentar as pré-condições e as pós-condições associadas a cada um dos métodos abordados no modelo, mediante a representação abstrata da biblioteca Qthread) para que verificadores baseados em BMC possam suportar a verificação de programas multi-tarefas Qt;

- Replicar o comportamento dos métodos analisados, com o intuito de verificar e validar se o comportamento do Modelo Operacional Simplificado Multi-Tarefas Qt imprime um comportamento similar ao do *framework Qt*;

- Realizar a verificação de propriedades específicas de cada biblioteca e classes, com o auxílio da inclusão de pré e pós-condições, com o auxílio da inclusão de asserções, contendo as as propriedades que devem ser satisfeitas;

- Criar um conjunto de suítes de teste, contendo pelo menos um caso de teste positivo e um caso de teste negativo, para a validação de todos os métodos desenvolvidos no modelo (e.g, uma vez que até o presente momento não existia nenhuma base de dados para a validação de programas multi-tarefas Qt);

- Avaliar o desempenho do modelo operacional que propomos com diferentes solucionadores baseados em SMT (Z3, Boolector, Yices);

- Avaliar a eficiência do modelo operacional que propomos mediante a sua utilização nos processos de verificações de três verificadores de modelos baseados em BMC (ESBMC, DIVINE, LLBMC).

Nesse sentido, primeiramente foram analisadas e catalogadas todas as bibliotecas responsáveis pela manipulação e implementação das estruturas de dados que são utilizadas na implementação de programas concorrentes no *framework Qt* . Serão ainda consideradas as bibliotecas e funções que necessitam ser re-implementadas devido a ocorrências de heranças múltiplas (funcionalidade ainda não suportada pelo ESBMC++ (RAMALHO et al., 2013)), que ocorrem com frequência no referido *framework*. Esta tarefa torna-se necessária para que possa haver o mapeamento de todas bibliotecas, classes e propriedades específicas que serão implementadas no modelo operacional que está sendo proposto. Com base nisso, pretende-se explorar a verificação de modelos utilizando o modelo operacional criado e adicionalmente incluir aplicações reais para testes e, por fim, documentar todos os resultados obtidos. Pretende-se, ainda, adicionar este modelo operacional ao verificador de modelos *ESBMC++* para que essas aplicações possam ser verificadas de forma automática.

1.3 Descrição da Solução

A abordagem proposta tem como objetivo que o modelo operacional criado seja uma extensão adicional para o verificador *Efficient SMT-based Context-Bounded Model Checker* ESBMC++ (RAMALHO et al., 2013), para o tornar apto a realizar a verificação de estruturas de dados que implementam a computação paralela e a concorrência presente no *framework Qt* . Sendo implementado fora do núcleo do *ESBMC++*, de forma que o modelo operacional seja passado para o *ESBMC++*, apenas no momento da verificação, este modelo operacional ajudará o *parsing* do ESBMC++ a identificar as estruturas do *framework Qt* que serão passadas a ele durante o processo de verificação. Dessa forma, o *parsing* do ESBMC++ será capaz de transcrever o código que estará sendo passado na entrada, em uma estrutura de dados chamada Árvore Sintática Abstrata (AST) (MAGERMAN, 1995), que conterá as informações que serão necessárias para a correta verificação do algoritmo.

Uma das principais soluções encontradas foi construir o modelo considerando que as *threads* executam de forma assíncrona, o que nos permite evitar a modelagem síncrona, que é extremamente dispendiosa (CORDEIRO, 2011).

No processo de modelagem, utilizamos uma estratégia de isolar as funções e os métodos alvos da verificação, sendo definidos de acordo com o nível de importância que a mesma desempenha no funcionamento do sistema, por meio da inclusão de pré e pós-condições. Focamos também na redução do tamanho das instâncias de problema do BMC, buscando uma análise mais profunda, considerando os recursos das ferramentas BMC que foram utilizadas para o processo de verificação, buscando, assim, diminuir o tempo de verificação.

Como já foi mencionado, quanto maior o número de instruções adicionadas ao código a ser verificado, maior será a quantidade de intercalações que serão geradas no decorrer da verificação, de tal forma que buscamos adicionar às intercalações de uma forma preguiçosa, uma vez que as restrições são adicionadas conforme é necessária a checagem de determinada propriedade,

reduzindo a profundidade da AST que será gerada durante o processo de verificação. Utilizando essa abordagem conseguimos diminuir a complexidade dos processos de verificações.

1.4 Contribuições

O presente trabalho apresenta quatro contribuições gerais:

- Primeiramente, foi desenvolvido um estudo para realizar a verificação de programas concorrentes *Qt*, fazendo uso de técnicas de verificação modelos.
- Segundo, aplicamos a verificação de modelos para verificação de programas multi-tarefas Qt, atrelada à técnica BMC baseada em SMT, mediante ao uso do verificador de modelos ESBMC++.
- Terceiro, desenvolvemos um modelo operacional que pode ser adicionado aos processos de verificações do verificador ESBMC++, tornando-o portável à verificação de programas multi-tarefas Qt.
- Quarto, provamos que o modelo proposto é portável a verificadores de modelos, que prestem suporte a verificação de programas multi-tarefas escritos em *C/C++*, para fins de validação da metodologia que propomos, utilizamos além do ESBMC outros dois verificadores de modelos presentes na literatura, são eles o verificador de modelos LLBMC (MERZ; FALKE; SINZ, 2012) e o verificador de modelos DIVINE (BARANOVÁ et al., 2017), os mesmos foram utilizados em conjunto com o modelo operacional simplificado multi-tarefas Qt, para a validação de um conjunto de suítes de testes contendo problemas de concorrência extraídos da literatura, que foram desenvolvidos e testados durante a construção da metodologia que propomos.

O trabalho é estendido de um trabalho anterior [ESBMCQtOM: A Bounded Model Checking Tool to Verify Qt Application] (MONTEIRO et al., 2017). Aqui, os aspectos de implementação e o seu uso não são discutidos, exceto pelo fato de que nessa ocasião incluímos no ESBMCQtOM novos recursos essenciais do módulo Qthread. Dessa forma, a principal contribuição desse trabalho é atender às principais funcionalidades do módulo Qthread presentes no framework Qt. Partindo dos resultados obtidos, foi desenvolvido um módulo para o ESBMCQTOM chamado Modelo Operacional Simplificado Multi-Tarefas Qt, capaz de identificar propriedades presentes no módulo Qthread do framework Qt.

1.5 Organização da Dissertação

A introdução apresenta os objetivos e motivações desta pesquisa, implementados durante o desenvolvimento deste trabalho. O restante das seções está dividido da seguinte forma:

O Capítulo 2: Fundamentação Teórica na qual se apresenta uma revisão dos conceitos básicos por trás da verificação formal de programas.

O Capítulo 3: Trabalhos Relacionados, onde se apresenta um resumo de diversas ferramentas e técnicas de verificação empregadas, para verificar programas concorrentes e provar corretude de tais programas utilizando a técnica *BMC*.

O Capítulo 4: Verificação de Programas Multi-Tarefas, em que se apresenta o processo de criação e instanciação da classe *Qthread* e como foi utilizada como base para a implementação de um modelo simplificado da biblioteca *Qthead* com o auxílio da utilização da biblioteca *POSIX*. Além disso, são registradas as regras de conexões e tratamento de explosão do espaço de estados que podem estar presentes em programas multi-tarefas Qt.

No Capítulo 5, apresentam-se os resultados da verificação dos casos de teste, desenvolvidos em *C++*, e a comparação dos resultados com as ferramentas, em que o modelo foi adicionado, são elas o *ESBMC* (CORDEIRO et al., 2012; MORSE et al., 2013; MORSE et al., 2014), *DIVINE* (MERZ; FALKE; SINZ, 2012; BARNAT et al., 2013) e *LLBMC* (FALKE; MERZ; SINZ, 2013).

Por fim, o Capítulo 6, apresenta as contribuições do trabalho, além de apresentar sugestões de pesquisas futuras.

2 Fundamentação Teórica

Este capítulo descreve os fundamentos que serão a base para a compreensão das técnicas descritas nos Capítulos seguintes. Em especial, serão descritos os conceitos principais para o entendimento da verificação formal, a arquitetura do verificador *ESBMC++* (CORDEIRO et al., 2012) e o compilador *LLVM* (LATTNER; ADVE, 2004) usados pelos verificadores de modelos *LLBMC* (MERZ; FALKE; SINZ, 2012) e *Divine* (BARNAT et al., 2013) em seus processos de verificações, e as características estruturais do módulo *Qt/C++*, que será abordado pelo presente trabalho. Tratar-se-ão, também, sobre as técnicas de verificação de modelos que serão utilizadas para tornar portável a verificação de estruturas concorrentes em *Qt*, a partir de verificadores de modelos que suportam estruturas de dados escritas em *C++*, que ofereçam suporte a biblioteca *POSIX* (Portable Operating System Interface, 2015).

2.1 Fundamentos Lógicos

2.1.1 Lógica Proposicional

A presente seção apresentará uma breve definição da Lógica Proposicional (LP), introduzindo a sua sintaxe, a semântica e alguns exemplos de implementação.

Lógica nada mais é do que o estudo das formas que são modeladas partindo de argumentos bem definidos extraídos de sistemas, sendo modeladas de forma abstrata, buscando atender às proposições que comprovam as premissas do argumento, sem preocupações do que as formas modeladas significam no mundo real, podendo ser modeladas a partir de um conjunto bem definido de símbolos e de regras de derivação (BRADLEY; MANNA, 2007).

A LP é amplamente utilizada na representação formal de um sistema, que deriva dos argumentos válidos. Os argumentos do sistema são modelados, a partir de certas proposições, que possuem as premissas do argumento, também conhecidas como assertivas. As assertivas representam uma lógica binária, que se baseia nas premissas de que uma determinada sentença só poderá assumir dois valores, ou é verdadeiro, representado pelo símbolo $\top$, ou é falso, representado pelo símbolo $\bot$, as variáveis proposicionais: $x_1,..,x_n$ (o conjunto de variáveis proposicionais na maioria das literaturas é denotado pela letra X e n corresponde a um número finito de variáveis proposicionais). Operadores lógicos (por exemplo, $\neg$,$\wedge$), também chamados de operadores booleanos, fornecem o poder expressivo da LP, de modo que a assertiva final é verdadeira, se sua conclusão seguir todas as premissas estabelecidas pelas premissas do argumento (CORDEIRO, 2011).

A representação das proposições pode ocorrer mediante a utilização de símbolos e de um sistema de regras, que representaram as proposições lógicas formadas a partir da combinação de proposições atômicas, considerando o uso de conectivos lógicos e sistemas de derivação, o que

permite que as fórmulas possam ser representadas em formato de teoremas do sistema. Ou seja, generalizando, a LP é formada a partir de uma proposição, que é formada por um conjunto de outras proposições.

Com isso, podemos dizer que a LP é uma lógica binária, que usa como principais premissas, o fato de que determinadas afirmações devem ser verdadeiras ou falsas, ou, ainda, que seja possível realizar uma demonstração de que uma conclusão é verdadeira dentro de um determinado contexto com um conjunto de hipóteses (proposições verdadeiras). Podemos assumir que a dedução ou a conclusão de uma assertiva, ocorre a partir de outras assertivas ou premissas de dedução, que nada mais são do que os argumentos válidos do sistema. É importante ressaltar que as deduções se apoiam em premissas anteriores, ou seja, algumas assertivas devem ser consideradas como as primeiras para não ficarmos presos em laços infinitos, é importante salientar que para a assertiva inicial, não é necessária uma dedução. Essas deduções são chamadas de axiomas na literatura (BRADLEY; MANNA, 2007). Partindo daí, os axiomas, também conhecidos como regras de inferência, são os argumentos válidos do sistema, representando os mecanismos que definem como as assertivas passam de uma para a outra, resultando assim em asserções, as quais são chamadas de teoremas de sistema.

Uma fórmula LP é definida em termos dos elementos básicos: Verdadeiro, falso, uma variável proposicional x ou a aplicação de um dos seguintes operadores lógicos em uma fórmula φ, dentre os operadores lógicos os mais utilizados são: "not"($\neg\ \varphi$), "and"($\varphi_1 \wedge \varphi_2$), "or"($\varphi_1 \vee \varphi_2$), "implica"($\varphi_1 \Rightarrow \varphi_2$), "Se e somente se"($\varphi_1 \Leftrightarrow \varphi_2$). "Disjunção exclusiva"($\varphi_1 \oplus \varphi_2$) ou "Expressão Condicional"(ite ((Θ, ϕ_1,ϕ_2))) (CORDEIRO, 2011).

O número dos argumentos das fórmulas em LP são previamente mensurados. Os operadores são unários(negação), binários(todos os outros operadores) e ternários("ite"). Já os operadores que estão do lado esquerdo do símbolo $\Rightarrow$ são chamados antecedente, já os que estão do lado direito, são chamados de consequente. Átomos ou proposições atômicas, são proposições indecomponíveis, são formados a partir de variáveis proposicionais e constantes proposicionais.

Definição 2.1 *As regras definidas para a sintaxe das fórmulas LP, são definidas pela gramática a seguir:*

Fml ::= Fml $\wedge$ *Fml* | $\neg$*Fml* | *(Fml)*| *Atom*

Atom ::= Variável | *Verdadeiro* | *Falso*

Definição 2.2 *A representação da LP pode ser feita simbolicamente com o uso de um alfabeto, que é definido por:*

1. *Símbolos de pontuação: ("e ");*

2. *Com o uso de um conjunto enumerável de símbolos proporcionais:* $p, q, r, \ldots$.

3. *Operadores e conectores lógicos:*

    - *De aridade zero:* $\top$*(constante que denota a proposição falsa);*

- *Binários:* $\wedge$ *(conjunção),* $\vee$ *(disjunção),* $\rightarrow$ *(implicação);*
- *Unário:* $\neg$ *(negação);*
- *Expressões condicionais : ite;*

Exemplos da utilização dos operadores lógicos em fórmulas (CORDEIRO, 2011):

- $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\, \varphi_1 \wedge \varphi_2\,)$
- $\varphi_1 \Leftrightarrow \varphi_2 \equiv (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$
- $\varphi_1 \oplus \varphi_2 \equiv (\varphi_1 \wedge \neg\, \varphi_2) \vee (\varphi_2 \wedge \neg\, \varphi_1)$
- ite $(\Theta,\, \phi_1{,}\phi_2) \equiv (\Theta \wedge \phi_1) \vee (\neg\, \Theta \wedge \phi_2)$

Definição 2.3 *Uma fórmula LP é dita uma fórmula bem formulada, quando a Definição 2.1 é utilizada durante a construção da fórmula.*

Definição 2.4 *Nós definimos a precedência relativa dos operadores lógicos do mais alto para o mais baixo, como segue:* $\neg$, $\wedge$, $\vee$,$\Rightarrow$ *e* $\Leftrightarrow$*. (CORDEIRO, 2011)*

Para a validação das fórmulas LP, inicialmente avaliamos se as variáveis proposicionais são verdadeiras ou falsas, tal validação ocorre a partir de interpretações. De tal forma que para cada variável proposicional é atribuída uma interpretação correspondendo a um valor verdade. Por exemplo, I = $x_1 \mapsto tt$, $x_2 \mapsto ff$, corresponde a uma atribuição verdadeira para x_1 e falsa para x_2. Onde, o valor verdade de uma fórmula em LP pode ser calculado com a utilização de indução matemática ou do uso da tabela verdade. Considerando as possíveis avaliações de uma variável proposicional x (isto é, tt ou ff), os operadores lógicos $\neg$, $\wedge$, $\vee$, $\Rightarrow$,$\Leftrightarrow$ e $\oplus$ podem ser expressos em uma tabela verdade, como mostrado na Tabela 1 (CORDEIRO, 2011).

Tabela 1 – Tabela Verdade

x_1	x_2	$\neg x_1$	$x_1 \bigwedge x_2$	$x_1 \vee x_2$	$x_1 \Rightarrow x_2$	$x_1 \Leftrightarrow x_2$	$x_1 \oplus x_2$
ff	ff	tt	ff	ff	tt	tt	ff
ff	tt	tt	ff	tt	tt	ff	tt
tt	ff	ff	ff	tt	ff	ff	tt
tt	tt	ff	tt	tt	tt	tt	ff

Na Tabela 1, é possível ver um exemplo de como modelar a LP. De acordo com os operadores utilizados na Definição 2.2, ambos os exemplos se equivalem. Podendo ser comprovado pela Definição 2.3.

Segundo Cordeiro (CORDEIRO, 2011), a partir de operadores básicos é possível descrever uma definição indutiva da semântica da LP que define o significado dos operadores básicos e de

fórmulas mais complexas. Onde podemos aferir que ϕ avalia para tt e I $\models \phi$ se ϕ avalia para ff sob I.

Definição 2.5 *Para análise de formas mais complexas a semântica pode ser avaliada da seguinte maneira:*

- $I \models \phi_1 \Rightarrow \phi_2$ *se, sempre que* $I \models \phi_1$ *então* $I \models \phi_2$
- $I \models \phi_1 \longleftrightarrow \phi_2$ *se,* $I \models \phi_1$ *e* $I \models \phi_2$*, ou* $I \not\models \phi_1$ *e* $I \not\models \phi_2$

2.1.1.1 Davis-Putnam-Logemann-Loveland(DPLL)

O núcleo dos BMCs modernos utilizam fórmulas lógicas para descrever os estados e as transformações que ocorrem entre as transições dos estados do sistema, com o auxílio de fórmulas LP para verificar a satisfatibilidade. Isso é possível mediante a utilização de solucionadores SMT, que podem ser utilizados para verificar a satisfabilidade de fórmulas booleanas, sobre uma ou mais teorias, utilizando a combinação de diferentes teorias de *background*. O solucionador Z3 (MOURA; BJØRNER, 2011) da Microsoft (BJØRNER; PHAN, 2014) , Boolector da FMV e o Yices (DUTERTRE, 2014) da SRI International (DENKER; KAGAL, 1978), usados neste trabalho em conjunto com o verificador de software ESBMC, são solucionadores que usam o DPLL e são classificados como completos, conforme Moura e Bjorner (2008), Gomes et al (2007) e Moura et al (2006).

São baseados em uma variante do algoritmo *Davis-Putnam-Logemann-Loveland(DPLL)* (BARRETT, 2013). O algoritmo DPLL é baseado no algoritmo *backtracking*, no processo de verificação da satisfatibilidade de fórmulas LP, de tal forma que, cada solucionador emprega sua técnica, para que a codificação de seus problemas sejam SAT compatíveis. (TRINDADE et al., 2015)

O algoritmo retorna satisfatível (SAT, do inglês *satisfiable*), quando os valores forem atribuídos para todas as variáveis e nenhum conflito for encontrado, ou retorna insatisfatíveis (UNSAT, do inglês *unsatisfiable*), caso contrário.

O Algoritmo 1 exemplifica o pseudocódigo que representa o funcionamento da técnica original usada pelo algoritmo DPLL, onde os parâmetros de entrada são dados a partir da Forma Normal Conjuntiva (fnc ou ou do inglês CNF) de um problema (S) e a profundidade (t) da árvore de busca (ou seja, o número total de todas as variáveis booleanas do problema). Onde o conjunto de variáveis são denominadas por P e variam de $P_1, P_2, P_3.....P_n$. De tal forma que

Algorithm 1 Algoritmo DPLL

```
 1: function DPLL(S)
 2:     if S = {} then
 3:         return SAT
 4:         if {} ∈ S then
 5:             return UNSAT
 6:         else
 7:             return l ← umliteralqualquerretiradodeS
 8:             if  thenDPLL(S \ t) = SAT
 9:                 return SAT
10:             else
11:                 return DPLL(S \ −t)
12:             end if
13:         end if
14:     end if
15: end function
```

Fonte: (TRINDADE et al., 2015)

o condicionamento CNF é aplicado ao processo de ramificação na variável abaixo da variável atual, representado por $S\mid_{d_1}$ ou no caso de negação $\urcorner S\mid_{d_1}$, isso vai depender da variável que está sendo analisada.

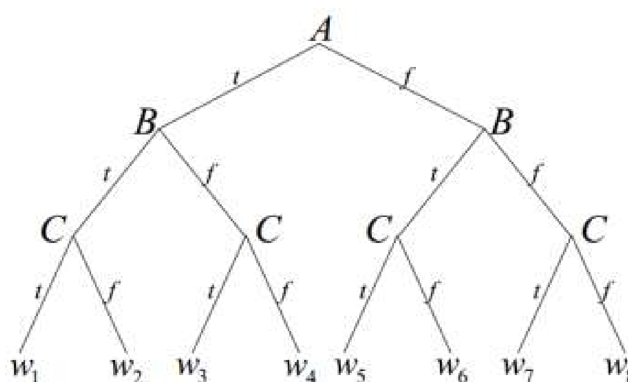


Figura 1 – Árvore de busca resultante da enumeração de todos os possíveis estados das variáveis A, B e C . (PIPATSRISAWAT; DARWICHE, 2009)

Na Figura 1, podemos observar a ilustração de uma árvore binária contendo três variáveis A, B e C. O literal t corresponde a verdadeiro (true) e f corresponde a falso (false), ramificados em variáveis de decisão. O valor atômico w_i corresponde ao nó da folha que representa um possível caminho na árvore.

2.1.2 Bounded Model Checking (BMC)

O *Efficient SMT-based Context-Bounded Model Checker* (ESBMC), é um verificador de modelos limitado ao contexto que se baseia nas teorias SMT, é usado para verificar originalmente programas ANSI-C (ALVES et al., 2018). No ESBMC o programa que será verificado é modelado por um sistema de transição de espaço de estados $M = (S, R, s_0)$, gerado a partir de um Fluxo de Controle do Programa (do inglês (*Control-Flow Graph* (CFG))(GADELHA et al., 2013), onde $s_0 \subseteq S$ representa o conjunto de estados iniciais do programa. Um estado $s \in S$ representa os correspondentes de todas as variáveis presentes no programa e no valor do contador de programa (ALVES et al., 2018). As transições de dois ou mais estados $\gamma(s_i, s_{i+i}) \in R$ representam um conjunto de transição (i.e., pares de estados do CFG, que determinam as transições entre os espaços de estados do sistema). A transição entre o estado atual s_i e o próximo estado s_{i+1} pode ser representada como $\gamma(s_i\ ,\ s_{i+1})$, os estados possuem as restrições dos valores das variáveis do programa e do contador do programa (CORDEIRO, 2011). De forma que durante o processo de verificação é gerado automaticamente um grafo de controle. Um nó no CFG representa uma atribuição (determinística ou não determinística) ou uma expressão condicional, enquanto que uma aresta representa uma mudança no fluxo do programa (GADELHA et al., 2013). Dado um sistema de transição M, uma propriedade φ e um limite k, o ESBMC desdobra o sistema k vezes e transforma o resultado em uma condição de verificação (VC, do inglês *verification condition*) ψ, de tal forma que ψ é satisfeita se, e se somente se, φ possuir um contra-exemplo de comprimento menor ou igual a k (GADELHA et al., 2013). A representação do problema envolvendo a técnica BMC pode ser expresso pela Equação 2.1.

$$\psi_k = I(s_0) \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{2.1}$$

Onde φ é uma propriedade, I é o conjunto de estados iniciais de M e $\gamma(s_i\ ,\ s_i\ +1\ )$ é a relação de transição de M entre os passos j e $j+1$. Logo, I(s_0) $\wedge \bigwedge_{j=0}^{i-1} \gamma\,(s_j\ ,\ s_{j+1})$ representa a execução de M, i vezes, e a Definição 2.4 só poderá ser satisfeita se, e somente se, para um $i \leq k$, existir um estado alcançável no passo em que ϕ é violada. Se a Equação da Definição 2.4 for satisfeita, então o ESBMC mostra um contra-exemplo, definindo quais os valores das variáveis necessárias para reproduzir o erro. O contra-exemplo para uma propriedade ϕ é uma sequência de estados s_0 , s_1 , ..., s_k com $s_0 \in S_0$ e $\gamma(s_i,\ s_{i+1})$ com $0 \leq$ i $<$ k. Se a Definição 2.4 não for satisfeita, pode-se concluir que nenhum estado com erro foi alcançável em k ou menos passos (CORDEIRO, 2011).

2.1.3 Efficient SMT-based Context-Bounded Model Checker

O ESBMC é um verificador de modelos limitado ao contexto que se baseia em teorias do módulo da satisfatibilidade (SMT), é utilizado para verificar programas C/C++ utilizando solucionadores SAT/SMT (GADELHA et al., 2018). Ele processa programas C utilizando o compilador Clang. Em especial, o código escrito em C é compilado em um programa *GOTO* que é equivalente ao programa original. Após passar por esse pré-processamento, o programa gerado em formato *GOTO* pode ser processado por uma máquina de execução simbólica.

O ESBMC realiza a computação das restrições utilizando duas funções, C e P, as quais computam as atribuições de variáveis e as checagens de propriedades, tais como condições de corridas e assertivas definidas pelo usuário. Adicionalmente, o ESBMC gera condições de verificações, por exemplo, estouros aritméticos, violação dos limites de um vetor e de referência de ponteiro nulo (*NULL pointer dereference*). Um gerador de VC (VCG, do inglês *Verification Condition Generator*) então deriva VCs a partir das propriedades (GADELHA et al., 2013).

2.1.4 Low-Level Bounded Model Checker(LLBMC)

O LLBMC (*Low-Level Bounded Model Checker*) é um verificador de programas C/C++ sequenciais, que aplica a técnica BMC utilizando solucionadores SMT (MERZ; FALKE; SINZ, 2012). Diferentemente do CBMC e do ESBMC, o processo de verificação não utiliza o código fonte do programa, mas uma representação intermediária gerada pelo compilador LLVM. A ferramenta é capaz de verificar diversos defeitos, entre eles *overflow* e *underflow* aritmético, divisão por zero, deslocamento inválido de bits, acesso ilegal de memória (por exemplo, violação de limite de vetor), além de assertivas inseridas pelo próprio desenvolvedor (GADELHA et al., 2013).

É possível observar na Figura 2 a abordagem que é utilizada pelo LLBMC com o intuito de verificar programas C/C++ (FALKE; MERZ; SINZ, 2013). Primeiramente o programa que será verificado é convertido em uma Representação Intermediaria LLVM (LLVM IR), mediante a utilização do compilador Clang (LATTNER; ADVE, 2004). O desdobramento dos laços até um valor é definido pelo usuário (ou pode ser determinado automaticamente) é realizado por meio de transformações que ocorrem diretamente na LLVM IR (GADELHA et al., 2013). Em seguida, a LLVM IR é transformada em uma Representação Intermediária Lógica (LLVM ILR), que estende a lógica QF_ABV (BARRETT et al., 2010) utilizando expressões para codificar as

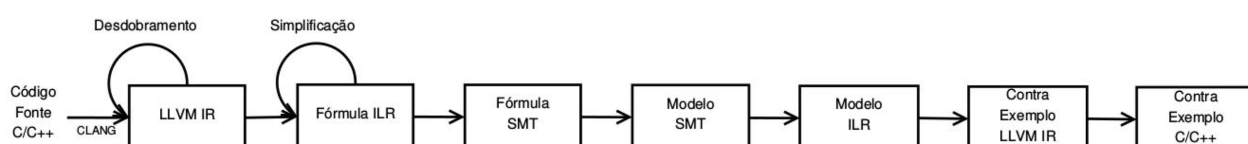


Figura 2 – Arquitetura do LLBMC (GADELHA et al., 2013).

verificações feitas pelo LLBMC (BARRETT, 2013).

Na próxima etapa, a fórmula ILR é simplificada utilizando um conjunto de regras de simplificação. Essas regras são suficientes para descartar várias expressões "simples", antes de serem passadas para o solucionador SMT. Ao final da simplificação, a fórmula ILR é transformada em uma fórmula SMT, que é solucionada utilizando o solucionador SMT STP (GANESH; DILL, 2007). Caso a fórmula seja satisfeita, é gerado um modelo SMT contendo o erro, que é então convertido em um modelo ILR, que é utilizado para construir um contra-exemplo em LLVM IR.

2.1.5 Técnicas de Verificação de Modelos

A quantidade de sistemas embarcados vem aumentando substancialmente nas últimas décadas (CORDEIRO, 2017). Estes sistemas têm se tornado cada vez mais complexos, o que gera um desafio adicional para os engenheiros e projetistas com o intuito de obter a segurança e confiabilidade no produto final desenvolvido (CORDEIRO; FILHO, 2016).

Dessa forma, a verificação de software tem desempenhado um papel importante no produto que é desenvolvido (GADELHA et al., 2018). Em especial, as empresas têm investido grande parte de seus recursos e esforços nos processos de verificação. Diversos *frameworks* têm sido usados com o intuito de acelerar o desenvolvimento desses produtos, é partindo desse ponto que o *framework* Qt representa um excelente conjunto de classes reutilizáveis, tornando possível desenvolver bibliotecas e aplicativos, podendo ser compilados em diversas plataformas sem a necessidade de alterar o código-fonte, podendo ser utilizado em ambientes desktop KDE e dispositivo móveis da Nokia, entre outros (MONTEIRO et al., 2017).

Para a obtenção da garantia da ausência de erros não previstos, a utilização de testes de software tradicionais podem ser inviáveis, seja por dificuldades financeiras ou técnicas, uma vez que pode ocorrer a influência de agentes externos que afetam no resultado final da verificação. Com isso, a verificação formal das propriedades necessárias para o funcionamento correto dos sistemas torna-se extremamente útil.

É nesse cenário que a verificação formal surge como uma técnica eficiente, no que diz respeito a validação de sistema embarcados críticos, buscando a máxima garantia de corretude. Essa técnica tem por objetivo, provar matematicamente a conformidade de um determinado algoritmo com relação a uma determinada propriedade, atendendo a métodos formais (BAIER; KATOEN, 2008; CLARKE; HENZINGER; VEITH, 2018).

A verificação formal deriva-se em verificação dedutiva e verificação de modelos. A verificação dedutiva ocorre com o auxílio de um conjunto de fórmulas que descrevem as propriedades dos sistemas, utilizando axiomas e regras de provas, funciona bem com espaços de estados finitos, muito embora seja difícil de realizar a prova de teoremas de forma automática. Em contrapartida, a verificação de modelos checa as propriedades de um sistema, executando exaustivas explorações de todos os comportamentos possíveis que o programa pode assumir durante o processo de execução, funcionando apenas em modelos de estados finitos, sendo usado fortemente para a verificação de propriedades de corretude de sistemas de espaços de estados finitos de forma completamente automática.

Na verificação de modelos, o sistema sob análise e as suas especificações são representados matematicamente, a partir de proposições lógicas (lógica temporal linear e de ramificação), de tal forma que é possível verificar se uma dada fórmula é satisfeita, de acordo com uma determinada propriedade (SOUSA, 2013).

Um grande desafio da verificação formal encontra-se no tratamento das explosões de espaços

de estados, que pode ocorrer caso o sistema contenha muitos componentes interagindo paralelamente, ou mesmo, caso a estrutura de dados assuma valores variados. Nesses casos, os estados globais crescem de forma exponencial com o número de processos. Para realizar a verificação de forma automática, verificadores de software, baseados em BMC, tem sido muito utilizados. Os verificadores BMC são em grande parte desenvolvidos tendo como base principal as teorias de verificação de modelos, tendo como objetivo principal realizar de forma confiável a checagem das principais propriedades do sistema.

Um verificador de modelos que tem se destacando é o verificador de modelos ESBMC. O verificador de modelos *ESBMC* é baseado no *front-end Clang/LLVM* para programas *C/C++*, suportando diferentes teorias e solucionadores SMT (GADELHA et al., 2018). Este explora informações de alto nível para simplificar e reduzir o tamanho das fórmulas geradas.

2.2 Tecnologias Multi-Tarefas no Qt

O *framework Qt* (The Qt Company Ltd., 2015) oferece várias classes e funções que oferecem suporte a *threads*, são elas:

- *QThread* é a base de todo o controle das *threads* no *Qt*. Cada instância *QThread* representa e controla uma *thread*. *QThread* pode ser instanciada diretamente ou com subclasses. A instanciação de uma *QThread* fornece um laço de eventos paralelos, permitindo que os *slots QObject* sejam invocados em uma *thread* secundária. A subclasse de uma *QThread* permite inicializar uma nova *thread* antes de iniciar seu laço de eventos ou para executar o código em paralelo sem um laço de eventos.

- *QThreadPool* e *Qrunnable* reutilizam as *threads*: criar e destruir as *threads* com frequência pode ser caro. Para reduzir essa sobrecarga, as *threads* existentes podem ser reutilizadas para novas tarefas. A *QThreadPool* é uma coleção de *QThreads* reutilizáveis. Cada aplicação *Qt* possui um conjunto de *threads* globais, que são acessíveis pela rotina *QThreadPool :: globalInstance()*. Esse conjunto de *threads* globais mantém automaticamente um número ideal de *threads* com base no número de núcleos na CPU. No entanto, uma *QThreadPool* separada pode ser criada e gerenciada explicitamente.

- *Qt Concurrent* (manipulação de *threads* em alto nível): O módulo Qt *concurrent* fornece funções de alto nível que lidam com alguns padrões comuns de computação paralela:

mapear, filtrar e reduzir. Ao contrário do uso de *QThread* e *QRunnable*, essas funções nunca exigem o uso de primitivas de *threading* de baixo nível, como mutexes ou semáforos.

- *WorkerScript*: (*Threads* no QML). O tipo *QML* do *WorkerScript* permite que o código *JavaScript* seja executado em paralelo com a *threads* da *GUI*.

Como enumerado acima, o *Qt* fornece diferentes bibliotecas para o suporte ao desenvolvimento de aplicativos multi-tarefas. A solução certa para um determinado aplicativo depende do objetivo da *thread* criada e da duração da sua execução.

Das bibliotecas disponíveis, a principal para o suporte a threads no Qt (criação, manipulação e finalização das *threads*), é a biblioteca de baixo nível *Qthreads*, que é herdada por todas as outras bibliotecas disponíveis para a manipulação das *thread* no QT. Sendo assim foi a biblioteca selecionada para ser abordada pelo presente trabalho, uma vez que a mesma servirá de base para o suporte futuro de outras bibliotecas suportadas pelo Modelo Operacional Simplificado Multi-Tarefas Qt que propomos.

Devido a sua natureza multi-plataforma, o Qt oculta todo o código necessário para realizar o gerenciamento das threads, uma vez que a plataforma pode ser executada em diferentes sistemas operacionais, e cada sistema operacional possui suas próprias rotinas de gerenciamento e manipulações das threads de usuário. Para que seja possível executar alguma rotina em uma thread é necessário subclassificar e substituir o método QThread::run(), o run() é chamado por padrão por todos os threads recém criados. Caso o método run(), não tenha sido reescrito, a implementação padrão do compilador do Qt chama por padrão o método Qthread::exec(), o exec() é o método responsável por inserir um thread no loop de eventos.

```
1 class Thread : public QThread {
2 protected:
3  void run() {
4   }
5 };
6
7  int main(int argc, char *argv[])
8 {
9     QCoreApplication app(argc, argv);
10     HelloThread thread;
11     thread.start();
12     thread.wait();
13     return 0;
14 }
```

Figura 3 – Exemplo de código para a criação de threads no Qt.

Por intermédio da biblioteca *Qthreads*, o *framework* Qt é capaz de fornecer mecanismos de baixo e de alto nível para a sincronização de *threads*, possibilitando assim o processo de exclusão mútua no Qt (The Qt Company Ltd., 2015). No *Qt*, cada *thread* tem sua própria pilha de execução, o que significa que cada *thread* tem seu próprio histórico de chamadas e variáveis locais. No diagrama da Figura 4 é possível ver como as *threads* estão localizadas na memória.

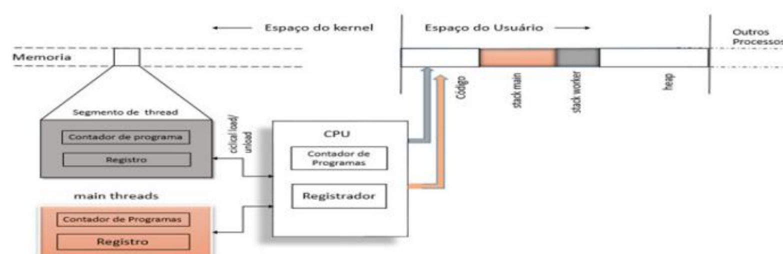


Figura 4 – Funcionamento da thread no framework Qt (COMPANY,) .

Os registros das *threads* inativas e os contadores de programas são tipicamente mantidos no espaço do *kernel*. Existe uma cópia compartilhada do código em uma pilha separada para cada *thread*. Programas que utilizam multi-tarefas em geral são difíceis de testar, em grande parte pela sua execução não determinística, o que potencialmente aumenta as explosões do espaços de estados.

2.2.1 Verificação de Sistemas Multi-Tarefas

Segundo Hansen (HANSEN, 1978), a programação concorrente proporciona a construção de sistemas de software que são compostos por um número fixo de programas sequenciais, os quais são executados concorrentemente. Tais sistemas de software são denominados concorrentes e os programas sequenciais que o compõem cooperam entre si para produzir um resultado ao final da execução do sistema. Durante a execução de um sistema de software concorrente, o processador intercala a execução dos programas sequenciais até que se alcance o resultado final da execução do sistema de software (SANTOS et al., 2016).

As técnicas de verificação para programas multi-tarefas tradicionais têm o principal objetivo de checar todas as possíveis intercalações que a *thread* pode realizar durante o processo de execução, tendo como custo computacional a sobrecarga do sistema, sem a garantia de que o sistema será completamente coberto durante o processo de verificação. Sistemas de software que

implementam multi-tarefas são difíceis de validar, devido a dois problemas que ocorrem durante o processo de verificação:

- Primeiro, o processo de execução dos programas multi-tarefa ocorre de forma não determinística;
- Segundo, durante o processo de verificação os espaços de estados gerados podem se tornar muito grandes;

Em contrapartida, a verificação de modelos para esse tipo específico de sistemas, pode garantir uma cobertura completa, mesmo que possivelmente exija um alto custo computacional, uma vez que possivelmente durante o processo de verificação será gerado um número exponencial de estados do sistema.

Na Seção 2.2, foram introduzidos os conceitos principais de sistemas de transições de estados e foi discutido o problema que é encontrado na verificação de modelos associada á técnica *BMC* para verificação de programas que possuem uma única *thread*. Já para a verificação de programas multi-tarefas, a noção de espaço de estados se mantêm, visto que os valores são atribuídos às variáveis de forma sequencial, com apenas uma diferença, para programas multi-tarefas, as intercalações de transições de estados são considerados para várias *threads* diferentes, uma vez que, em um programa multi-tarefas, existem diversas *threads* executando em paralelo, tendo sua execução agendada de forma não determinística por um planejador de sistema global, como será explicado na próxima seção.

Por enquanto, nós assumimos informalmente que o comportamento operacional das threads que executam em paralelo é dado pelos sistemas de transição M_1,. . ., M_n (recorda definição 2.14). Conforme Cordeiro (CORDEIRO, 2011), podemos então definir um sistema de transição $M_t = \cup_n^{j=0} M_j$ que específica o comportamento da composição paralela dos sistemas de transição M_1 a M_n.

Esta seção descreve os principais mecanismos usados para modelos de sistemas multi-tarefas representados por sistemas de transição de estados, que são basicamente compostos por *N threads* individuais, o que permite codificar de forma explícita os modelos de intercalações que estão implementados na estrutura do *BMC*, para possibilitar a verificação automática de programas multi-tarefas. Mais informações podem ser encontradas, nas referências disponíveis na literatura tais como (MCMILLAN; HWANG, 1990; PRABHU et al., 2011).

2.2.2 Modelagem de Sistemas Multi-Tarefas em Qt

Um programa de software Qt multi-tarefas a ser analisado é modelado como uma tupla $M = S, S_0, T, V$, onde :

- S é um conjunto de estados e $S_0 \subset S$ o conjunto de estados iniciais;
- $T = t_0, t_1, ..., t_n$ representa o conjunto das *threads* que entram em execução no programa, n representa a quantidade de *threads* em execução;
- $V = V_{global} \cup V_j$ onde V_{global} é o conjunto de variáveis globais e V_j é o conjunto de variáveis locais de t_j.

Assumimos que cada variável pode possuir um tamanho finito de domínio. O estado $s \in S$ consiste nos valores das variáveis locais e globais da linguagem Qt. Assumimos que cada *threads* j é uma tupla $t_j = (R^j l^j)$, onde;

- $R^j \subseteq S$ é a relação de transição das *threads* t_j;
- $l^j = \langle l_i^j \rangle$ é a sequência de localização das *threads* l_i^j, no tempo de espera de i.

2.2.3 Concorrência e Intercalação

Existem dois modos de execução simultânea; assíncrono e síncrono (QADEER, 2008). No modo assíncrono, consideramos que apenas uma *thread* pode entrar em execução por vez, enquanto no modo síncrono todas as *thread* podem ser executadas em paralelo. As *thread*, no modo assíncrono, podem se comunicar por intermédio de troca de passagens ou de variáveis compartilhadas.

No modelo de transmissão de mensagens, as *thread* podem enviar/receber mensagens (incluindo zero ou mais *bytes*, estruturas de dados ou até segmentos de código) para/de outras *thread*. No modelo de variável compartilhada, uma região de memória pode ser acessada simultaneamente por várias *thread* para fornecer uma comunicação entre elas.

A sincronização ou serialização das *threads* (por exemplo, por exclusão mútua ou variável de condição) garante que várias *threads* não acessem regiões específicas da memória simultaneamente. Isso significa que, se uma *thread* tiver iniciado o acesso a uma região da memória,

qualquer outra *thread* que tente acessar essa região deverá aguardar até que a primeira *thread* seja concluída.

Este trabalho considera programas multi-tarefas no modo assíncrono e assume que as threads em execução no modelo só se comunicam por variáveis compartilhadas e executam em uma área específica da memória de forma sincronizada, para evitar o acesso simultâneo as variáveis compartilhadas. Observe que essa suposição também se aplica à verificação de software em sistemas com vários núcleos, pois a operação assíncrona é uma solução padrão para evitar a violação de memória em processadores com vários núcleos (DEJNOZKOVÁ; DOKLÁDAL, 2004).

Um paradigma amplamente adaptado para programas multi-tarefas é o da intercalação.

Neste paradigma, uma sequência de intercalação representa uma execução possível do programa onde todos os eventos concorrentes são organizados em uma ordem linear. Assim, a noção de simultaneidade é representada por intercalações, ou seja, a escolha das threads que entram em execução ocorre de forma não determinística entre as múltiplas *threads* que estão em execução simultânea. Essa perspectiva é baseada no fato de que apenas um núcleo está disponível, no qual as ações das *threads* são intercaladas.

Do ponto de vista da modelagem, esse conceito também se aplica se as *threads* forem executadas em diferentes núcleos. Em ambos os casos (*single-core* ou *multi-core*), existem muitas sequências de intercalação com diferentes ordenações entre eventos simultâneos. A representação das intercalações de concorrência de dados depende essencialmente do "escalonamento"do sistema operacional, que durante o processo de execução das *threads* é o responsável pela etapa de intercalação simultânea das *threads*, obedecendo a estratégia definida pelo algoritmo do "escalonador". Atualmente, alguns algoritmos de escalonamento têm sido muito utilizados, os mais conhecidos são:o FIFO(First in, first out) (RU; KEUNG, 2013), SJF (Shortest Job First) (RU; KEUNG, 2013), SRT (Shortest Remaining Time) (SCHRAGE, 1968),RR (Round-Robin) (SHREEDHAR; VARGHESE, 1996), Múltiplas Filas (ZAHARIA et al., 2009), Algoritmo Fair-Share (KAY; LAUDER, 1988). Esse tipo de representação abstrai completamente a velocidade das *threads* participantes e, portanto, modela qualquer realização possível por uma máquina de núcleo único ou por vários núcleos com velocidades arbitrárias (CORDEIRO, 2011).

Considerando o cenário da verificação de modelos, com o intuito da verificação completa das especificações dos programas concorrentes, torna-se necessário considerar todas as sequências de interações que as *threads* em execução pode assumir. Isso pode resultar em espaço de estados

extremamente grande que deve ser explorado por um verificador de modelos, que, por sua vez, é a principal fonte do problema de explosões de estados (CORDEIRO, 2011).

Um exemplo desse processo de execução, pode ser observado se considerarmos um Grafo de Fluxo de Controle(CFG) que representa duas *threads*, a primeira *thread* e representada por T_a e a segunda é representada por T_b. Como pode ser visto na Figura 5 , onde as variáveis, a e b são declaradas como variáveis globais. Para cada T_i, seu grafo de fluxo de controle é representado por um grafo direcionado, que pode ser expresso pela fórmula T_i=$\langle$ N_i, E_i,n_{io} $\rangle$, onde N_i corresponde ao conjunto de nós de instruções do programa, E_i corresponde ao conjunto de arestas que representam transições do sistema. Em nosso exemplo, a *thread* T_a=$\langle$ N_a, E_a, n_{a0} $\rangle$, os nós correspondem a N_a={ T_{a0}, T_{a1}, T_{a2}, T_{a3} }, as arestas correspondem a E_a=($T_{a0} \rightarrow T_{a1}$, $T_{a1} \rightarrow T_{a2}$, $T_{a2} \rightarrow T_{a3}$) e o nó inicial corresponde a n_{a0}=T_{a0}, enquanto isso, na *thread* T_b=$\langle$ N_b, E_b, n_{b0} $\rangle$ os nós correspondem a N_b={ T_{b0}, T_{b1},T_{b2},T_{b3} }, as arestas correspondem a E_b=($T_{b0} \rightarrow T_{b1}$, $T_{b1} \rightarrow T_{b2}$, T_{b2} } T_{b3}) e o nó inicial corresponde a n_{b0}=T_{b0}.

Dizemos que uma instrução de programa é visível se acesa uma variável global, caso contrário, é invisível. Em nosso exemplo, consideramos que todas as declarações do programa (ou seja, $a = 2$, $a = a + (b/3)$, $b = 6$ e $b = b + 3$) são visíveis. Um entrelaçamento representa uma possível execução do programa, em que todos os eventos simultâneos são organizados em uma ordem linear. Qualquer alteração em uma *thread* ativa, em uma transição no grafo é considerada como uma mudança de contexto.

Uma declaração de programa é considerada atômica se nenhuma mudança de contexto puder acontecer durante sua execução. Instruções que envolvem no máximo uma variável global não são afetadas por mudanças de contexto. Em nosso exemplo, as declarações de programa $a = 2$, $b = 6$ e $b = b + 3$ são atômicas, enquanto a declaração do programa $a = a + (b/3)$ não são atômica, porque são afetada por trocas de contexto.

O CFG que representa todas as sequências possíveis de intercalação das *threads* T_a e T_b. O número de sequências de intercalações possíveis para um dado número de *threads* N consistindo de s instruções em um programa sem laços pode ser calculado como segue (SANGIOVANNI-VINCENTELLI et al., 2004).

$$I = \frac{(\sum_{i=1}^{N} s_i)!}{(\prod_{i=1}^{N}(s_i!))} \tag{2.2}$$

Em nosso exemplo de execução, temos $N = 2$, $s_a = 2$ e $s_b = 2$ e o número de sequências de

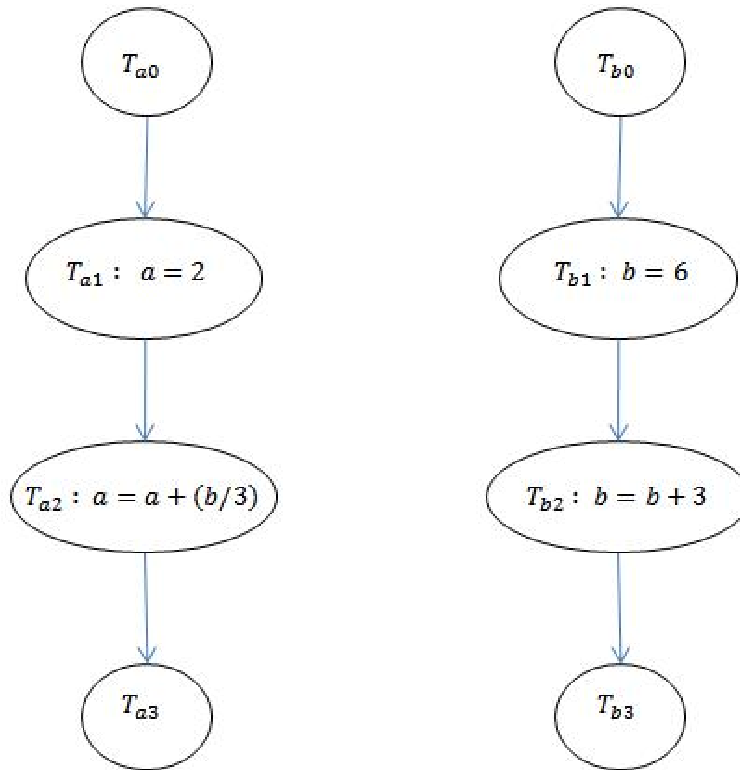


Figura 5 – Grafo de Fluxo de Controle Direcionado .

intercalação possíveis é:

$$I = \frac{(2+2)!}{2! * 2!} = \frac{24}{6} = 6 \tag{2.3}$$

O sistema de transição que representa a execução paralela das *threads* TA e TB.

2.3 Resumo

Neste capítulo, exploramos, inicialmente, as teorias básicas para a verificação de modelos de programas multi-tarefas a partir do uso de ferramentas BMC. Abordamos, preliminarmente, os fundamentos da lógica proposicional, que é a base pela qual se fundamenta a verificação de modelos, definimos a sintaxe da gramática, os operadores e operações lógicas, bem como as suas

regras de precedências, extração de predicados para a modelagem das proposições e teoremas usados para a definição de pré e pós-condições, que são aplicadas para a detecção de possíveis violações de propriedades.

Em seguida, apresentamos uma breve introdução sobre a teoria do módulo de satisfabilidade, exploramos um pouco a notação padrão da teoria, bem como sua aplicação, a partir da utilização de solucionadores SMT, que decidem a satisfabilidade de fórmulas lógicas extraídas a partir de fórmulas lógicas bem definidas, abordamos a técnica BMC aplicada aos verificadores de modelos limitados ao contexto ESBMC e DIVINE, baseados em teorias SMT para verificar programas C/C++.

Apresentamos a técnica de verificação de modelos, que corresponde à ideia base da apresentação da nossa proposta, exploramos a sua aplicação e vantagens no processo de verificação de sistemas multi-tarefas.

Adicionalmente, apresentamos uma descrição sobre o *framework* de desenvolvimento Qt para o desenvolvimento de sistemas multi-tarefas Qt, exploramos a forma com a qual são realizados os acessos simultâneos aos dados realizados pelas *threads*, exploramos uma breve introdução das funções disponibilizadas pelo *framework* Qt para manipulações das *threads*, bem como suas limitações em comparação com a biblioteca base para todas as bibliotecas de controle de *threads* no Qt, denominada QThread, que é a biblioteca que abordamos no trabalho.

Como resultado, o conteúdo explorado neste capítulo subsidia o embasamento necessário para a compreensão do trabalho que propomos na presente pesquisa, que será abordado mais detalhadamente nas próximas seções.

3 Trabalhos Relacionados

No campo de verificação de modelos, não é conhecida a existência de verificadores de software capazes de realizar a verificação de tais estruturas do *framework Qt*. Sendo assim, inviabilizado a comparação com outros verificadores. Entretanto, diversas ferramentas realizam a verificação de software mediante ao uso da técnica BMC. A mesma tem se popularizado, tendo como motivação principal o crescente surgimento de sofisticados solucionadores SMT (MOURA; BJØRNER, 2008). Nesse campo, a ferramenta *Low-Level Bounded Model Checker* (LLBMC) é um bom exemplo. A ferramenta LLBMC adota a técnica BMC no processo de verificação de programas escritos em ANSI-C/C++. Essa ferramenta utiliza o compilador LLVM para converter programas ANSI-C/C++ na representação intermediaria LLVM, que, por sua vez, perde informações sobre a estrutura dos respectivos programas em C++ (i.e., as relações entre classes) [Monteiro et al. 2015]. De maneira semelhante, o ESBMC também faz uso de solucionadores SMT (MOURA; BJØRNER, 2008), para checar as condições de verificação geradas a partir de programas escritos em C++. No entanto, de forma oposta à abordagem que é proposta no presente trabalho, o LLBMC não oferece suporte ao tratamento de exceções, o que dificulta a verificação de aplicações desenvolvidas, em C++ (e.g., programas que dependem do conjunto de bibliotecas Standard Template Libraries – STL) (GADELHA et al., 2013)

Blanc et al. descrevem a verificação de programas C++ que utilizam o conjunto de bibliotecas *Standard Template Libraries* (STL), via abstração de predicado [Blanc et al. 2007]. O modelo operacional proposto neste trabalho propõe que, utilizando-se tipos abstratos de dados, seja possível a verificação das bibliotecas STL e não a utilização da própria STL. Blanc et al. expõe que a verificação do modelo operacional e suficiente para a verificação da corretude dos programas, tendo em vista que a prova das pré-condições, a partir de determinadas operações do modelo, são suficientes para que se tenha a garantia das referidas pré-condições que são especificadas pela linguagem alvo das operações. Essa abordagem mostra-se eficiente na busca por erros triviais em programas escritos em C++, no entanto encontra dificuldades quando são realizadas buscas mais profundas de erros, isso ocorre com frequência quando se trata da modelagem interna dos métodos inerentes à linguagem-alvo. No presente trabalho, esses problemas são contornados, com a simulação dos comportamentos que são utilizados por determinadas funções e métodos.

3.1 Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking

Cordeiro e Fischer at al. (CORDEIRO, 2011), descrevem e tratam de três abordagens de verificação de modelos para programas multi-tarefas com variáveis compartilhadas e operações de bloqueio, usando a Verificação de Modelos Limitados(BMC) (BIERE et al., 2003) baseadas em Teoria do Módulo de Satisfatibilidade (SMT) (MOURA; BJØRNER, 2008).

- A primeira abordagem de verificação de modelos é a preguiçosa, a partir dela é possível gerar todas a possíveis intercalações que serão executadas pelos sistemas durante o processo de verificação e executar os solucionadores SMT (MOURA; BJØRNER, 2008) em cada umas das intercalações executadas pelo programa.

- A segunda abordagem de verificação de modelos utilizado, chamadas de gravação de escalonamento, codifica todas as intercalações possíveis em uma fórmula única e explora a rapidez dos solucionadores utilizados.

- A terceira abordagem de verificação de modelos utilizada é a aproximação e ampliação, essa abordagem reduz o número espaços de estados, abstraindo, assim, o número de intercalações das provas de insatisfabilidade geradas pelos solucionadores SMT (MOURA; BJØRNER, 2008).

Assim como no modelo operacional simplificado multi-tarefas Qt que propomos, as primitivas de sincronização da biblioteca Pthread também são usadas para modelar as *threads*. O verificador de modelos ESBMC durante os seus processos de verificações produz um programa instrumentado. Com relação ao programa original, as intercalações produzidas durante o processo de verificação é limitada ao número de trocas de contexto que serão executadas durante o processo de verificação, o que o ajuda a diminuir o número de espaços de estados que são gerados no decorrer do processo de verificação, o que diminui significativamente o tempo de verificações.

Analisando os resultados alcançados pelo trabalho, as abordagens utilizadas para a verificação de programas multi-tarefas pelo ESBMC mostraram-se muito eficientes no processo de verificação e validação de sistemas concorrentes, uma vez que a abordagem utilizada foi capaz de detectar erros de atomicidade, erros de violação de ordem e bloqueios fatais e globais.

Analisando as três abordagens de acordo com os resultados alcançados a abordagem preguiçosa foi a que obteve os melhores resultados, sendo capaz de verificar todos os *benchmark* que foram propostos.

3.2 Verifying Multi-threaded C Programs with SPIN

Zaks et al. (ZAKS; JOSHI, 2008) propõem um verificador de modelos uma para verificar programas multi-tarefas usando o verificador de modelos SPIN (HOLZMANN, 1997). A ferramenta proposta na pesquisa compila um programa multi-tarefa em C, em um formato de *bytecode*, em sequência utiliza uma máquina virtual para interpretar os *bytecode* gerados e calcular os novos estados do programa através da ferramenta de verificação de modelos SPIN. A máquina virtual, é capaz de fornecer suporte à alocação dinâmica de memória e a biblioteca pthread. A abordagem proposta pode verificar o código após otimizações do compilador, que às vezes, podem introduzir condições de corrida. A ferramenta conta com um recurso que permite ao usuário definir funções de abstração de dados e restrinjam o número de trocas de contexto durante o processo de verificação. A partir da utilização do método *on-the-fly*, implementa um método de redução que o permite reduzir o número de trocas de contextos durante o processo de verificação. O processo de verificação ocorre da seguinte forma:

- Primeiramente, o processo de verificação da ferramenta apresentada na pesquisa ocorre por intermédio do verificador de modelo SPIN (HOLZMANN, 1997) que, por meio da conversão de um modelo PROMELA (junto com uma propriedade LTL a ser verificada) em um programa C (pan.c), que o codifica em um verificador de modelo, o qual verifica a propriedade em questão (de certa forma, portanto, SPIN é realmente um gerador de verificador de modelo) (ZAKS; JOSHI, 2008).

- Segundo, como o SPIN compila modelos desenvolvidos em C, versões recentes (desde o SPIN 4.0) permitem que fragmentos de programas C sejam incorporados aos modelos PROMELA. Cada um desses fragmentos é tratado como uma transição atômica determinística. Isso permite que o SPIN seja usado para verificar programas C em relação às especificações LTL.

- Terceiro, a criação do programa livre de exceções inicia com a geração de um grafo de fluxo de controle, denominado modular *interprocedural exception control-flow graph* ou

IECFG. O IECFG é, então, analisado por um algoritmo desenvolvido pelos autores que modela o conjunto das possíveis exceções que podem se conectar a blocos *catch* utilizando uma representação compacta, chamada Signed-TypeSet.

A máquina virtual foi projetada para ser usada com o SPIN, e a ferramenta resultante oferece suporte a quase todos os recursos do SPIN, como verificação de estado de bit e operação com vários núcleos. Também mostramos que resolvemos o problema de explosão de estado, permitindo que os usuários especifiquem funções de abstração, limites de trocas de contexto usando um algoritmo rápido para reduzir alternâncias de contexto desnecessárias (HOLZMANN; BOSNACKI, 2007).

3.3 Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization

Omar et al. (INVERSO et al., 2014) propões uma nova técnica BMC (BIERE et al., 2003) para programas multi-tarefas em C. Baseada em sequencialização, trata-se de uma ideia proposta por Qadeer e Wu (QADEER; WU, 2004). No trabalho apresentado é descrita uma nova abordagem para o BMC de programas C multi-tarefas para *threads*, modeladas a partir do padrão POSIX. Na abordagem proposta, o processo de verificação ocorre da seguinte forma:

- Primeiro, traduz-se um programa C multi-tarefas em um programa C sequencial não determinístico, que preserva a acessibilidade para todos os agendamentos *Round-robin*(um algoritmo empregado pelo escalonador de processos) com um limite determinado no número de rodadas.

- Segundo, reutiliza as ferramentas BMC(Lazy-CSeq (INVERSO et al., 2014), BLITZ (CHO; D'SILVA; SONG, 2013), CBMC (ALGLAVE; KROENING; TAUTSCHNIG, 2013), ESBMC (CORDEIRO, 2011) e LLBMC (FALKE; MERZ; SINZ, 2013)) de alto desempenho existentes como *back-end* para o problema de verificação sequencial. A tradução foi projetada para introduzir sobrecargas de memória muito pequenas para que produza fórmulas SAT/SMT (MOURA; BJØRNER, 2008).

A técnica proposta de sequencialização foi aplicada com sucesso na ferramenta Lazy-CSeq (INVERSO et al., 2014), a ferramenta possibilita a verificação:

(i) Das principais partes da API de *thread* do POSIX (Portable Operating System Interface, 2015), como criação e exclusão de *threads* dinâmicas e sincronização via variáveis , bloqueios e condições de corrida de *thread*.

(ii) A ferramenta oferece suporte a linguagem C de forma completa, como diferentes tipos de dados, alocação dinâmica de memória e recursos de programação de baixo nível, como aritmética de ponteiros.

Com o que foi exposto pode-se comprovar a viabilidade verificação de modelos a partir da utilização da técnica de sequencialização, pela ferramenta de protótipo Lazy-CSeq. O Lazy-CSeq também pode ser usado como um verificador de modelo independente uma vez que pode ser incorporado a outras quatro ferramentas BMC diferentes como *back-end*.

3.4 Efficient Modeling of Concurrent Systems in BMC

Ganai et al. (GANAI; GUPTA, 2008), propõe um método eficiente para modelar sistemas simultâneos multi-tarefas com variáveis compartilhadas e bloqueios no BMC (Bounded Model Checking) (BIERE et al., 2003). O método foi utilizado de forma eficiente para melhorar a detecção de propriedades de segurança, como corridas de dados. Na modelagem proposta por Ganai (GANAI; GUPTA, 2008):

- Primeiro, são criados modelos independentes (não acoplados) para cada *thread* individualmente no sistema;

- Segundo, os modelos criados são adicionados a variáveis e restrições de sincronização adicionais, de forma incremental e apenas onde essa sincronização é necessária para garantir a semântica de concorrência (com base em consistência sequencial).

A consistência sequencial é a semântica de simultaneidade mais comumente usada para desenvolvimento de software devido à facilidade de programação.

Com isso, o método propõe uma abordagem similar ao da pesquisa apresentada na presente dissertação para o gerenciamento dos *threads*, Ganai et al. (GANAI; GUPTA, 2008) concentra-se

principalmente na redução do tamanho das instâncias de problema do BMC para permitir uma busca mais profunda dentro dos recursos limitados, tempo e memória, visto que as *threads* são modeladas de forma síncrona, o que, de acordo com os dados levantados no trabalho é uma semântica de intercalação menos dispendiosa.

3.5 Verifying Atomicity Specifications for Concurrent Object-Oriented Software Using Model-Checking

Flanagan e Qadeer et al. (HATCLIFF; DWYER et al., 2004), propuseram declarações de atomicidade como um mecanismo leve para especificar propriedades de não interferência em linguagens de programação simultâneas, como Java, e forneceram um sistema de tipo e efeito para verificar as propriedades de atomicidade. Embora a verificação das especificações de atomicidade por meio de um sistema de tipo estático tenha várias vantagens (escalabilidade, verificação composicional), mostramos que a verificação por modelo também possui várias vantagens (menos anotações não verificadas, maior cobertura dos idiomas Java, verificação mais forte). Em particular, mostramos que, ao adaptar o verificador de modelo Bogor, naturalmente abordamos várias propriedades que são difíceis de verificar com um sistema de tipo estático.

O trabalho apresentado indica a existência de vários benefícios na verificação das especificações de atomicidade usando a verificação de modelo quando se emprega um sofisticado mecanismo de verificação de modelo dedicado ao software, como Bogor (FLANAGAN; GODEFROID, 2005). O verificador de modelos Bogor fornece um suporte avançado a linguagens orientadas a objetos para verificação de modelo, incluindo reduções de simetria de pilha, *garbage collection*, estratégias de redução parcial de ordem (POR) baseadas em escape estático e dinâmico, analisa e bloqueia as *threads*, e algoritmos sofisticados de compactação de estado que exploram o compartilhamento de estado do objeto.

Hatcliff e Dwyer et al. (HATCLIFF; DWYER et al., 2004), chegaram a conclusão de que sistemas de tipos e verificação de modelo são abordagens complementares para verificar a especificação de atomicidade de Flanagan e Qadeer (JR, 1977). Além disso, em verificadores de modelos como Bogor e JPF que fornecem suporte direto a objetos, a verificação de especificações de atomicidade deve ser incluída porque essas especificações são úteis, e a verificação de modelos

fornece um mecanismo de verificação eficaz para essas especificações.

3.6 Checking Concise Specifications For Multithreaded Software

Nas últimas décadas, várias técnicas baseadas em análise de fluxo de dados, demonstração de teoremas e verificação de modelos surgiram para a análise de software sequencial. No entanto, essas técnicas ainda não permitiram a verificação de sistemas de software grandes e multi-tarefas (FREUND; QADEER, 2004). E justamente uma solução para esse problema que Freund e Qadeer et al. (FREUND; QADEER, 2004) tentam apresentar em sua pesquisa: uma proposta de nova técnica de verificação de modelos para verificar propriedades de programas multi-tarefas. A análise realizada é baseada em sistemas com um número grande de operações e *threads*.

A partir da análise modular das *threads*, anotando cada variável compartilhada por um predicado de acesso, que resume a condição sob a qual uma *thread* pode acessar essa variável compartilhada, obtiveram uma análise modular de procedimento, anotando cada procedimento com uma especificação relacionada à sua implementação, isso é possível pelo usa da relação de abstração que combinam as noções de simulação e redução. Puderam implementar uma análise no Calvin-R, um verificador estático para programas Java multi-tarefas.

A ferramenta proposta verifica a modularmente se cada método em um programa satisfaz corretamente os predicados do sistema, que são extraídos a partir da abstração de sua especificação. Para cada verificação, o Calvin-R:

- Primeiro: Constrói um programa sequencial capturando os requisitos de correção necessários;
- Segundo, verifica se esse programa não está errado usando as técnicas de verificação existentes para programas sequenciais, entre elas estão:
  (i) Condições de verificação (FLANAGAN; SAXE, 2001; DIJKSTRA et al., 1976)
  (ii) provas de teoremas automáticos Simplify (NELSON, 1981).

O Calvin-R fornece uma forma para os programadores especificarem e verificarem propriedades complexas de programas multi-tarefas de maneira simples e intuitiva. Os predicados de acesso utilizados pelo Calvin-R permitem que os mecanismos de sincronização sejam expressos de

maneira simples e uniforme. Além disso, a relação de abstração baseada em simulação e redução permite a abstração de dados e controle nas especificações do método. Muitas propriedades do código multi-tarefas podem ser expressas de forma concisa e verificadas com essa técnica.

3.7 Resumo

No presente capítulo as características inerentes aos trabalhos relacionados foram estabelecidas, elencadas e comparadas. Nas subseções, foram apresentadas as ferramentas e metodologias de destaques na verificação de programas usando Modelos Operacionais para validação de programas concorrentes multi-tarefas.

SPIN (HOLZMANN; BOSNACKI, 2007) propõe o desenvolvimento de um Máquina Virtual capaz de fornecer suporte a alocação dinâmica de memória para a manipulação de *threads* para programas multi-tarefas desenvolvidos em C++, usando uma abordagem de abstração de funções e restrição do número de mudanças de contextos. Lazy-CSeq foi desenvolvido como um modelo operacional para programas multi-tarefas desenvolvidos em C, com a utilização de um modelo simplificado da biblioteca Pthread, traduzindo um programa C multi-tarefas em um programa C sequencial não determinístico, para garantir o correto funcionamento do modelo proposto foi desenvolvido um escalonador do tipo Round-robin(um algoritmo empregado pelo escalonador de processos desenvolvido) com um limite determinado de trocas de contextos. É importante salientar que a mesma metodologia foi empregada para garantir a escalabilidades entre as *threads* no decorrer do processo de desenvolvido no Modelo Operacional Simplificado Multi-Tarefas Qt ,apresentado na pesquisa que propomos. O Lazy-CSeq utilizou em seus processos de testes os verificadores LLBMC e ESBMC, os quais também foram utilizados para a validação da metodologia proposta na presente pesquisa. Borgor e Calvin-R são ferramentas de verificação de modelos que foram utilizadas por Modelos Operacionais Abstratos para a verificar programas multi-tarefas Java. Os modelos adicionados aos verificadores Borgor e Calvin-R, utilizam uma estratégia similar a que foi empregada no processo de desenvolvimento do modelo operacional que propomos. Quando comparados à metodologia usada para o desenvolvimento do modelo que propomos na nossa pesquisa com os modelos desenvolvidos e adicionados aos verificadores Borgor e Calvin-R, é possível verificar que também é utilizada a estratégia de inclusão de pré-condições, pós-condições e proposições atômicas, que são adicionadas ao longo do desenvolvimento dos modelos, com o fim de validar que as propriedades inerentes às

Tabela 2 – Tabela de resultados- Verificadores de modelos

Ferramenta	**Metodologia**	**Linguagem**	**Solucionador**	**Suporte a Lib Pthread**	**Suporte a Orientação a Objeto**	**Verif. propriedades Atômicas**	**Processo Automatizado**
ESBMC	BMC e redução de ordem parcia	C e C++	3, Boolector e Yices	Sim	Sim	Propriedades verificas por asserções	Sim
SPIN	Análise estática e redução de ordem parcial	C	Sim	Sim	Não	Propriedades analisadas por pré e pos condições	Sim
Lazy-CSeq	Modelo Operacional Simplificado	C	Sim	Sim	Não	Propriedades analisadas por pré e pos confições	Sim
Bogor	Modelo Operacional Abstrato	Java	Sim	Não	Sim	Propriedades analisadas por pré e pos confições e validação de propriedades	Sim
Calvin-R	Modelo Operacional Abstrato	Java	Sim	Não	Sim	Propriedades analisadas por pré e pos confições e validação de propriedades	Sim

linguagens que estão modeladas, estão sendo corretamente atendidas.

=

4 Verificação de Programas Multi-Tarefas Qt

Neste capítulo, descrevemos e avaliamos um modelo operacional para estruturas de dados concorrentes do *framework* Qt, com o intuito de possibilitar a verificação de software multi-tarefas com variáveis compartilhadas e de bloqueios, usando verificadores de modelos limitados ao contexto com base no Módulo das Teorias da Satisfabilidade (SMT). Em especial, realizamos a modelagem das *threads* a partir da utilização de primitivas de sincronização existentes na biblioteca padrão do POSIX (Portable Operating System Interface, 2015), a qual representa um conjunto de documentos produzidos pela IEEE e adotado pelo ANSI e ISO (International Organization for Standardization, 2015), que abrange desde a interface básica dos sistemas operacionais, até questões de administração e segurança (CORRÊA; FRIEDRICH, 1997). Esta modelagem é suportada pelos verificadores de modelos utilizados para a validação do modelo operacional que propomos, o que nos permitiu realizar a simulação da comunicação entre processos, compartilhamento de memória, sincronização e escalonamento das *threads* nos verificadores BMC utilizados. Para tanto, realizamos a modelagem das *threads* em dois níveis: a primeira é a nível de sistema (as *threads* são escalonadas umas em relação a todas as outras) e a segunda é a nível de processo (as tarefas são escalonadas com relação as *threads* pertencentes ao mesmo processo) (CORRÊA; FRIEDRICH, 1997).

Nossos experimentos mostram que o ESBMC pode analisar problemas maiores e reduzir substancialmente o tempo de verificação em comparação com técnicas em estado da arte que usam algoritmos iterativos de troca de contexto ou refinamento de abstração guiada por contraexemplo. Na construção do modelo nos baseamos na abstração de predicados, que correspondem a funções booleanas $P : X \rightarrow \{verdadeiro, falso\}$, de tal forma que, as proposições resultantes das abstrações, devem resultar em duas inferências lógicas (verdadeira ou falsa), dependendo do valor atribuído a suas assertivas, buscando abstrair ao máximo os componentes da linguagem. Tal abstração resultou em uma descoberta de erro mais rápida do que a verificação completa do código original, considerando que ao abstrair a linguagem alvo da verificação, o compilador consegue verificar a corretude de sistemas complexos de forma mais rápida, considerando a corretude das proposições as quais são necessárias à validação, buscando preservar o comportamento do sistema, considerando, ainda, que a simplificação das estruturas acarreta uma redução significativa dos estados durante a verificação (BAO; JONES, 2008). É importante ressaltar que

seguimos rigorosamente as especificações do código-fonte, fornecidos pelo *framework Qt* (The Qt Company Ltd., 2015).

4.1 Introdução

As técnicas de verificação baseada em BMC são amplamente utilizadas para a verificação bem sucedida de sistemas de *software* e para a detecção de erros em tempo de execução. Em especial, o BMC é utilizado essencialmente para realização de testes de falsificação, preocupando-se prioritariamente com possíveis violações de propriedades de lógicas temporais (LTL) (GODEFROID, 1996). É importante salientar que a ideia principal da técnica BMC (aplicada aos verificadores utilizados) não é provar a corretude de sistemas, mais sim verificar se uma determinada propriedade, inserida no modelo na forma de pré e pós-condições(expressas na forma de asserções), estão sendo violadas em uma profundidade pré determinada ϕ, em um limite de profundidade (*bound*) k.

Na tentativa de tratar a crescente complexidade dos sistemas de *software* que vem aumentando exponencialmente, os solucionadores de satisfabilidade booleana (SAT), que originalmente baseavam-se em diagramas de decisão (BDDs) (MCMILLAN, 1993) para modelar a representação simbólica de sistema, estão sendo substituídos por solucionadores das teorias do módulo da satisfabilidade (SMT), no processo de validação das condições de verificação (CVs) que são geradas durante o processo de verificação (GANAI; GUPTA, 2006). Um bom exemplo da utilização dessa metodologia é o proeminente verificador de modelo ESBMC (MORSE et al., 2014), que faz uso de técnicas de verificação simbólica de modelos baseadas em SMT, com um processo de verificação totalmente automático, possibilitando, assim, a verificação de programas sequenciais e multi-tarefas.

O principal desafio aqui é fazer com que o modelo criado tenha um comportamento similar ao das estruturas concorrentes presentes na linguagem alvo da verificação. No entanto, duas observações importantes podem nos ajudar:

- Qadeer e Rehof (QADEER; REHOF, 2005), apresentaram um método para identificar falhas em sistemas concorrentes, onde a análise do sistema é realizada com um número limitado de trocas de contexto. De tal forma puderam provar que a verificação de sistemas concorrentes torna-se decidível se a análise é realizada com um número limitado de trocas

de contexto, onde a quantidade máxima de trocas de contextos que podem ser realizadas durante o processo de análise dá-se por uma constante determinada de forma arbitrária. Conseguiram identificar com a metodologia proposta as falhas dentro do limite de trocas de contextos que foi estipulado previamente. Puderam provar que, a partir da limitação de trocas de contextos, não são necessárias mais de duas trocas de contextos para expor a falha em programas concorrentes, que tenham pelo menos dois *threads* em execução(QADEER; REHOF, 2005).

Com isso, puderam determinar que a maioria das falhas com relação a concorrência em aplicações reais são consideradas superficiais, de modo que apenas algumas mudanças de contexto são necessárias para expô-los (QADEER; REHOF, 2005).

- Segundo, de acordo com Cordeiro et al. (CORDEIRO; FISCHER; MARQUES-SILVA, 2012) , os solucionadores SAT e SMT produzem núcleos de insatisfatibilidade que nos permitem remover a lógica que não é relevante para uma dada propriedade (MCMILLAN, 2007). No nosso trabalho, buscamos usar tal conceito com o auxílio da inclusão de verificadores de modelos que ofereçam suporte a tais solucionadores.

Portanto, usamos durante o processo de verificação do modelo operacional proposto, uma análise limitada por contexto (LAL; REPS, 2009) que limita o número de chaveamentos (trocas de contextos) a serem explorados com o verificador de software ESBMC. Durante as verificações estipulamos o número máximo de trocas de contextos <*bound*> em 8, o que foi suficiente para expor as falhas durante o processo de verificação com o ESBMC, o que diminuiu significativamente o tempo de verificação dessas estruturas.

4.2 Conceitos Preliminares

No paradigma de intercalação amplamente adotado para programas multi-tarefas, a noção de simultaneidade é representada pela intercalação, ou seja, a escolha não determinística entre as *threads* em execução (MERZ, 2000). Caso um processador esteja disponível, as ações das *threads* que estão em execução devem ser intercaladas para o uso do processador. Esse conceito também é aplicável à múltiplos processadores, pois podem ocorrer muitas ordenações diferentes entre *threads* simultâneos. Essas intercalações representam uma possível execução do programa,

em que todos os eventos simultâneos são organizados em uma ordem linear. Qualquer alteração da *threads* em execução, que gere uma intercalação, é chamada de trocas de contexto.

As intercalações entre as *threads* em execução no programa ocorrem por intermédios de um escalonador, que utiliza algumas estratégias para o agendamento das *threads* que entram em execução no processador. Com isso, para que seja possível realizar a verificação de sistemas multi-tarefas, é necessário considerar todas as possíveis intercalações que as *threads* podem assumir durante o processo de execução, o que pode resultar em um grande aumento de espaços de estados, que serão explorados pelos verificadores de modelos utilizados durante o processo de verificação.

4.2.1 Programas Qt Multi-Tarefas

Durante a construção do modelo operacional, consideramos que programas multi-tarefas executam no modo assíncrono e assumimos que todas as *threads* comunicam-se por intermédio de variáveis compartilhadas.

Lembrando que, um programa multi-tarefa é uma lista (numerada) de comandos. Os comandos incluem atribuições não determinísticas ($V_{ar} = *$), tais como, instruções de bloqueio (para eliminar caminhos de execuções subsequentes) e instruções de assertivas (*assert* para indicar propriedades especificadas pelo usuário e pela linguagem alvo da verificação).

Todas as estruturas de controle são representadas por desvios condicionais para uma instrução $l \in \{1,...,n\}$. Uma *thread* t é uma sub-lista de comandos, que se divide entre a *thread* inicial e a *thread* final. As *threads* iniciais são criadas por meio de chamadas de procedimentos assíncronos, que retornam um valor inteiro que pode ser usado como o identificador da *thread* durante o processo de sincronização (*join* dá *thread*). Desta forma, a criação dinâmica da *thread* é permitida (CORDEIRO, 2011).

O verificador de modelos ESBMC, usa uma linguagem mínima semelhante à linguagem *GOTO* interna do verificador de modelos CBMC (CLARKE; KROENING; LERDA, 2004), para modelar programas multi-tarefas. A Figura 6 mostra um exemplo do funcionamento de um programa multi-tarefas representado em *Qt* e sua representação na linguagem *GOTO* multi-tarefas.

Neste exemplo de execução, temos três threads $thr1$, $thr2$ e $start$. Cada *thread* pode passar um ou mais argumentos como parâmetro, isto é, instruções que podem influenciar diretamente os

```
1 class Mythread : public
     QThread{
2   public:
3     Mythread(int s);
4     void run();
5     void thr1();
6     void thr2();
7     int num=0 ;
8 };
9 Mythread::Mythread(int s): num
     (s)
10 {};
11
12 void Mythread::thr1(){
13   num++;
14   if (num>1)
15     num = num-1;
16   return NULL;
17
18 };
19 void Mythread::thr2(){
20   bool y;
21   num ++;
22   y = num>1;
23   if (y)
24     num = num-1;
25   return NULL;
26 };
27
28 int main(int argc, char *argv
     []){
29  Mythread thr1,thr2;
30  thr1.start();
31  thr2.start();
32  thr1.thr1();
33  thr2.thr2();
34  thr1.wait();
35  thr2.wait();
36  assert(thr1());
37 };
38          (a)
```

```
1 int num = 0;
2 begin_thread thr1;
3   num = num+1;
4   if !(num>1)then goto L16;
5   num = num-1;
6 L14 : end_thread;
7 begin_thread thr2;
8  _Bool aux;
9  num = num+1;
10 aux = num > 1;
11 if !(aux)then goto L25;
12   num = num-1;
13 L23 : end_thread;
14 id1 = start_thread t1;
15 id2 = start_thread t2;
16 join_thread id1;
17 join_thread id2;
18 assert(num==1);
19 return 0;
20     (b)
```

Figura 6 – Exemplo de três *threads* executando simultaneamente o método *run()*.

estados do programa. Lembrando que as únicas instruções que causam alterações são atribuições e assertivas, já que os testes de fluxo de controle não podem influenciar o estado das instruções.

No exemplo da Figura 6(a), o método *start()* é o responsável por iniciar a execução das tarefas que a *thread* irá executar (o método re-implementável *run()* é executado por padrão sempre que o método *start()* é executado, a opção de implementá-lo fica a critério do desenvolvedor, porém sua implementação é recomendada pela documentação do Qt (The Qt Company Ltd. Documentation,

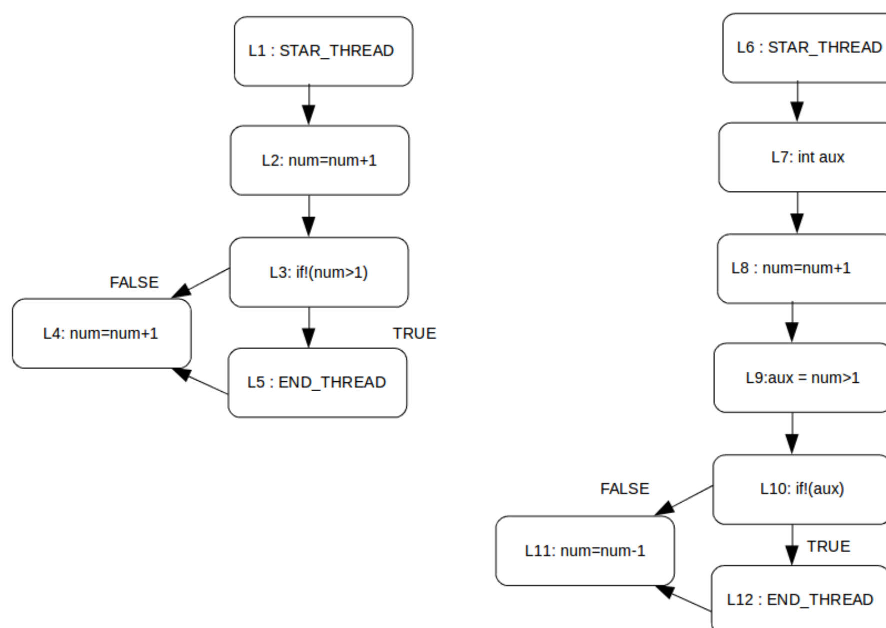


Figura 7 – Grafo de Fluxo de controle do programa em formato GOTO de duas threads

.

2015) para a manipulação das *threads* de sistema). O sistema operacional agenda a execução das *threads*, de acordo com os parâmetros de prioridades que são atribuídos ao método *start()*, no momento da execução.

No caso do exemplo da Figura 6(a), as prioridades das *threads* iniciadas são passadas por *default* às *threads* $thr1$ e $thr2$ (nas linhas 30 e 31), que são inicializadas pelo método *start()*. Como não foram atribuídas prioridades às *threads*, o escalonador executará uma *thread* por vez em ordem arbitrária.

No exemplo da Figura 6(b), a *thread* $thr1$ contêm duas instruções (nas linhas 3 e 5), a *thread* $thr2$ possui três instruções (nas linhas 8, 9, 10 e 12), enquanto a *thread* principal tem apenas uma (na linha 18). No nosso exemplo, um inteiro é atribuído como parâmetro para as *threads* $thr1$ e $thr2$, por intermédio do método *start()*.

Na Figura 7 é possível observar o fluxo de controle de execução do programa multi-tarefa convertido em GOTO exposto no Exemplo 6, para o fluxo de execução do thr1, inicialmente em L1 a *thread* é iniciada. Em seguida em L2 são realizadas as atribuições numéricas, em L3 é realizada uma verificação condicional, se *"num"* for menor que 1 o fluxo de controle termina o processo de execução, caso contrário, *"num"* é incrementado. Para o fluxo de execução do thr2, L6 inicializa a *thread* thr2, em L7 a variável *"aux"* e instanciada, em L8 *"num"* é incrementado, em L9 *"aux"* recebe um resultado *booleano* da comparação de *"num>1"*, em L10 é realizada

uma verificação condicional, se a atribuição for verdadeira o fluxo de controle é finalizado em L12, caso contrário L11 executa um decremento em L11.

4.3 Verificação de Modelo Limitado por Contexto de Software Qt Multi-tarefas

Para possibilitar a utilização da técnica de verificação de modelos, para programas multi-tarefas escritos usando o *framework* Qt, considerou-se inicialmente a sua complexidade. Utilizamos um modelo simplificado contendo apenas as estruturas que serão utilizadas durante a verificação das propriedades necessárias para a validação do correto funcionamento de cada um dos métodos abordados na presente pesquisa. Esta simplificação mostrou-se a melhor abordagem, uma vez que trata-se de um *framework* robusto, com um conjunto de bibliotecas padrões que contêm uma estrutura hierárquica grande e complexa, com um amplo conjunto de classes, módulos para manipulação de clientes e servidores, entre outros, o que tornaria a utilização de tais bibliotecas durante o processo de verificação uma abordagem extremamente difícil (SOUSA, 2013).

Deve-se salientar que o analisador sintático do *front-end* do *ESBMC* não é capaz de analisar todas as estruturas presentes no *framework Qt*, o que tornaria inviável a sua verificação (RAMALHO et al., 2013).

Deste modo, com o objetivo de fazer o *ESBMC* reconhecer estruturas do *framework Qt* que implementam o paralelismo e a concorrência de dados, foi necessário desenvolver a representação dos módulos simplificados da biblioteca *Qthread* (The Qt Company Ltd. Documentation, 2015) responsável por implementar e manipular as *threads* no *framework* Qt. Sendo assim, houve a necessidade do desenvolvimento de um modelo operacional simplificado que atenda a tais estruturas, ao qual chamamos de *Modelo Operacional Simplificado Multi-Tarefas Qt*, desenvolvido completamente em C++.

Com isso tornou-se possível diminuir a complexidade da AST que é produzida pelo *front-end* do ESBMC, diminuindo, assim, o custo computacional no processo de verificação.

Na Figura 8, é possível visualizar a inclusão do modelo no processo de compilação do ESBMC. A caixa em cinza representa o Modelo Operacional Simplificado Multi-Tarefas do Qt,

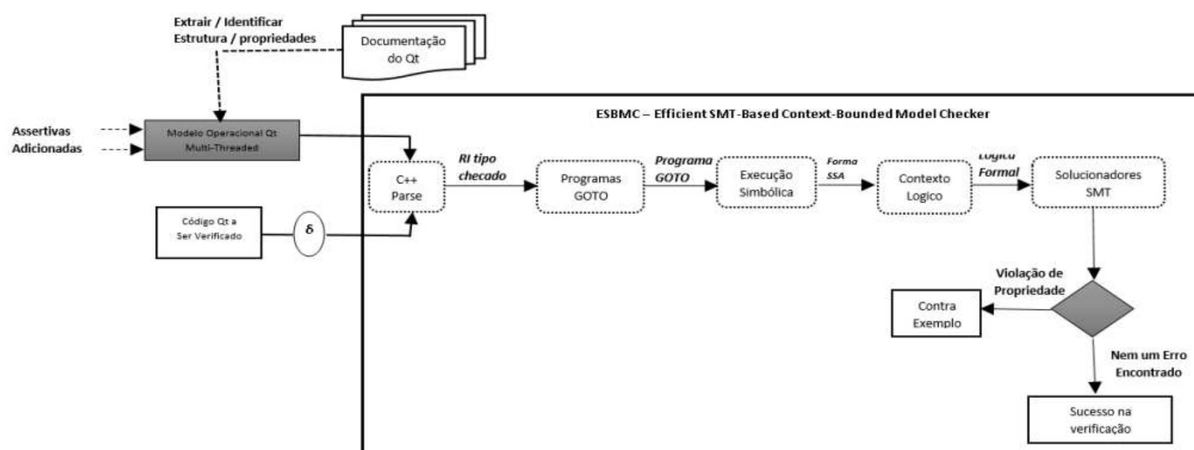


Figura 8 – Descrição do funcionamento do Modelo Operacional Multi-Tarefas Qt.

as caixas pontilhadas correspondem ao processo de execução do *ESBMC* e as caixas brancas representam as entradas e saídas.

Primeiramente o modelo operacional simplificado multi-tarefas Qt e o programa que será verificado são incorporados ao processo de compilação do verificador *ESBMC*, tal junção é possível pela utilização uma diretiva de pré-processamento (*i.e.*, *include*) que é adicionada na linha de execução do ESBMC, esse comando está representado na Figura 8 por δ, este contém as bibliotecas [1] do modelo operacional simplificado multi-tarefas Qt, os limites de tempo [2] e de memória [3]. É importante salientar que os parâmetros de tempo e memória foram definidos e ajustados de forma empírica.

No processo de verificação com o *ESBMC*, primeiramente o programa a ser verificado passa pelo analisador (*parsing*) do ESBMC, i.e., em seguida, é convertido em uma árvore de representação intermediária (do inglês, *Intermediate Representation* – IRep), posteriormente, a árvore de representação intermediária é convertida em um programa *GOTO* (por exemplo, expressões do tipo *if* e *while* são substituídas por expressões *GOTO*), o qual é executado de maneira simbólica pelo componente *GOTO-symex* implementando no ESBMC, tendo como resultado a criação de uma atribuição estática única (do inglês, *Single Static Assignment* - SSA) (CLARKE, 2003),

[1] -I /libraries/Containers/QThreads
[2] –memlimit 40000000
[3] –timeout 500

onde são consideradas todas as classes associadas do programa, incluindo atributos, assinaturas de métodos e afirmações.

Tendo como base a *AST*, o ESBMC realiza uma execução simbólica do programa que está sendo verificado. Como resultado desta etapa, o ESBMC gera equações *SMT* para restrições (atribuições e premissas de variáveis) e propriedades (condições de segurança), onde são posteriormente verificadas por solucionadores *SMT*. Deste modo, caso alguma violação de propriedade seja detectada durante o processo de verificação, o ESBMC retorna um contraexemplo relatando a linha em que o erro foi encontrado, as propriedades que foram violadas e as etapas que foram executadas durante a execução. Caso contrário, o *ESBMC* retorna que a verificação foi bem sucedida.

O modelo operacional multi-tarefas foi desenvolvido tendo como base a documentação original do *framework* Qt, sendo consideradas as estruturas de cada biblioteca e classes, incluindo atributos, assinaturas de métodos e protótipos das funções. Partindo da simplificação de tais estruturas, assertivas foram adicionadas ao modelo operacional multi-tarefas com intuito de garantir que cada propriedade seja formalmente verificada.

A Figura 9, demonstra o processo de desenvolvimento do Modelo Operacional Simplificado Multi-Tarefas Qt. A partir da documentação disponibilizada pelo *framework* Qt para a biblioteca QThread, identificamos as estruturas (métodos, funções e classes) necessárias para a execução correta do fluxo principal (criação, manipulação, bloqueio, finalização e etc) da biblioteca QThread. Em seguida, analisamos as propriedades da biblioteca que precisam ser atendidas para o funcionamento correto das estruturas presentes na biblioteca Qthread, isso é possível tendo como premissa a análise das prés e pós-condições que precisam ser atendidas durante o processo de execução das estruturas de dados desenvolvidas no modelo proposto. A validação do funcionamento correto das estruturas desenvolvidas no modelo dar-se pela da inclusão de assertivas contendo as pré e pós-condições que precisam ser atendidas pelos métodos e funções da biblioteca Qthread, assim podemos validar que as propriedades presentes nas estruturas de dados da biblioteca Qthread estão sendo executadas e atendidas de forma correta pelo modelo operacional que propomos e pelas estruturas de dados que são verificadas.

A partir das assertivas, o *ESBMC* é capaz de verificar e detectar violações relacionadas ao uso dos métodos, funções, pré-condições e pós-condições adicionados ao modelo.

É importante ressaltar que a metodologia proposta baseia-se no fato de que o Modelo Operacional Simplificado Multi-Tarefas Qt representa de maneira correta as bibliotecas aqui

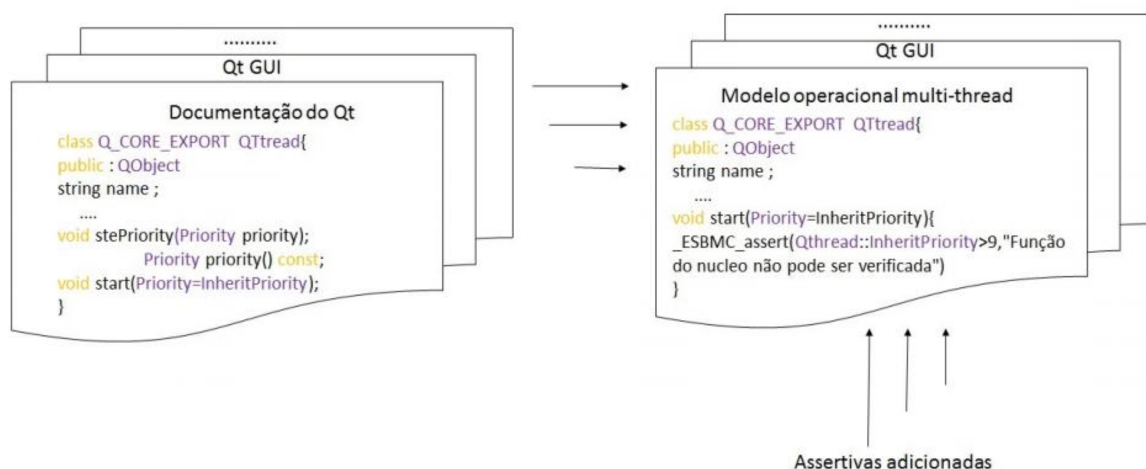


Figura 9 – Processo de desenvolvimento do Modelo Operacional Multi-Tarefas QT.

representadas do *framework* Qt (The Qt Company Ltd. Documentation, 2015), o que será mais explorado na Seção 4.4.1.

Dessa forma, todos os módulos desenvolvidos no presente trabalho foram testados manualmente e comparados com a biblioteca original *QThread*, com o intuito de garantir que o modelo desenvolvido possua um comportamento similar ao apresentado pela biblioteca responsável pela concorrência e o paralelismo de tarefas no *framework* Qt.

Isolando e definindo as pré e pós-condições dos módulos implementados, tornou-se possível uma validação abrangente das funcionalidades a serem validadas. Em especial, esta abordagem torna o processo de verificação por simplificação de predicados extremamente atraente, tendo em vista que este processo foca apenas nas funções e métodos alvos da verificação, tendo como auxílio a utilização de testes de unidade, onde cada método e seus dependentes foram testados individualmente, por meio de pequenos programas que executavam as funcionalidades presentes em cada um dos métodos incorporados nos módulos implementados do *framework Qt*.

Com o uso da técnica de simplificação de predicados, tornou-se possível reduzir expressivamente as explosões de espaços de estados, reduzindo, assim, a complexidade dos processos de verificações.

Dessa forma, torna-se possível a verificação de tais estruturas, uma vez que a biblioteca de sistema *Qthread*, não está disponível em *C++*, mas pode ser verificada por intermédio da criação de uma biblioteca abstrata, utilizando a biblioteca de sistema *POSIX/Pthread*, que é uma biblioteca de suporte a *threads* nativa do *C*. Em virtude disso, foi possível a reprodução do

comportamento dos módulos alvos da verificação, o que tornou possível reproduzir o comportamento de criação e a manipulação das *threads* de sistema do Qt, ao qual chamamos de Modelo Operacional Simplificado Multi-Tarefas Qt.

Com isso, mediante a utilização do modelo operacional simplificado multi-tarefas Qt que propomos, os verificadores de modelos baseados em BMC(ESBMC, DIVINE e LLBMC) puderam reconhecer e verificar, as estruturas presentes nas suítes de testes desenvolvidas para a validação de programas multi-tarefas Qt. É importante salientar que as suítes de testes não são apenas programas de usuários, elas incluem todas as bibliotecas de sistema que são necessárias para a execução dos casos de testes, incluindo as necessárias para a verificação das propriedades de sistema. Em virtude disso, todos os casos de testes foram escritos inteiramente em *C++*, a fim de tornar possível a validação das suítes de teste pelos verificadores de modelos utilizados nos processos de validação.

De um modo geral, o modelo operacional proposto mostrou-se portátil, considerando que foi possível adicioná-lo aos processos de análise e compilação de três verificadores (ESBMC, LLBMC e DIVINE) que utilizam a técnica de verificação baseada BMC em seus processo de validação e que ofereçam suporte à programas multi-tarefas escritos em *C++*. Ou seja, a adição do modelo os capacita a realizar o processo de verificação destas estruturas, sem mudar o comportamento do sistema a ser verificado, considerando o pressuposto que apenas adicionamos novas funcionalidades aos verificadores BMC.

4.3.1 Especificação de Propriedade

Um aspecto importante de um verificador é a especificação das prioridades que precisam ser validadas pelo modelo, com o intuito de garantir a corretude de todas as etapas durante o processo de verificação formal. Em nosso modelo, essa tarefa é atribuída às assertivas que são adicionadas ao modelo para validar todas as propriedades presentes nos módulos da linguagem alvo da verificação. O objetivo é provar que todas as intercalações que o processo de verificação executará não levará à um estado inválido, isto é, provar a preservação das propriedades de sistema em cada um dos possíveis eventos que podem ocorrer.

No modelo que propomos, o conceito de refinamento está ligado a ideia de incrementabilidade. Quando iniciamos a construção do modelo ele consistia de forma geral de uma versão mais simples e abstrata da biblioteca Qthreads, contendo apenas os construtores padrões. No entanto,

com a evolução do modelo novos detalhes foram adicionados a ele, buscamos garantir a cobertura de todos os requisitos definidos na fase de planejamento do modelo por intermédio da análise do comportamento das estruturas do framework Qt (tendo como referência a sua documentação (The Qt Company Ltd. Documentation, 2015)) e da validação do comportamento do modelo, o que foi possível com o uso da IDE Qt Creator.

Inicialmente as análises foram realizadas a partir da observação do comportamento dos módulos do Qt, o que foi possível mediante a utilização da ide do Qt Creator, para validar que o modelo operacional simplificado multi-tarefas Qt emprega o mesmo comportamento dos módulos implementados do Qt, utilizamos um conjunto de *benchmarks*:

- Primeiro, os *benchmarks* foram desenvolvidos e testados na idei do Qt, em seguida analisamos e salvamos o comportamento impresso pelo processo de execução.
- Segundo, os *benchmarks* foram executados no modelo com o auxílio do GCC, com o intuito de validar o comportamento das entradas e saídas das suítes de teste, uma vez que isso não seria possível com os verificadores de modelo BMC.

Com isso, foi possível corroborar que o modelo operacional que propomos emprega um comportamento similar ao apresentado pela ide do Qt, o que será abordado com mais detalhes na Seção 5.5. De tal forma que, durante o processo de criação do modelo, novas versões foram sucessivamente implementadas, cada uma delas correspondendo a uma versão anterior. No entanto, é importante salientar que não temos como garantir que tudo foi coberto, uma vez que não podemos validar as operações que ocorrem internamente no compilador do Qt, pois não somos capazes de reproduzir em sua totalidade o comportamento do compilador do Qt.

Entre tanto, apesar das limitações do modelo operacional que propomos, somos capazes de oferecer provas de validações para todas as funções que geram saídas visíveis aos usuários (o que nos permitiu a detecção de falhas de concorrência em tempo de execução), tais provas de validações são possíveis pelo do uso de variáveis globais compartilhadas. Mediante a análise do comportamento das variáveis globais é possível detectar e verificar possíveis violações no decorrer dos processos de verificações. Com isso, durante o processo de levantamento dos métodos, funções e classes que seriam abordados no modelo, consideramos apenas aqueles que nos permitiriam tal validação.

Em particular, uma importante propriedade que necessita ser validada é a correspondência entre os refinamentos do modelo e seus respectivos modelos abstratos. Essas validações ocorrem

por provas de validação, que são geradas durante o processo de desenvolvimento do modelo, como pode ser visto na Figura 10, onde é possível verificar as fases de desenvolvimento do modelo operacional.

As provas de validação são construídas a partir de um conjunto de propriedades atômicas, que são construídas na forma de predicados, baseados nas propriedades alcançadas pelo modelo, como pode ser visto no exemplo da Figura 11. Este exemplo mostra a criação dos objetos de classe *thrd1* e *thrd2* na linha 9; os objetos podem ser validados pelas assertivas que são utilizadas para a validação das instâncias dos objetos nas linhas 11, 13 e 15.

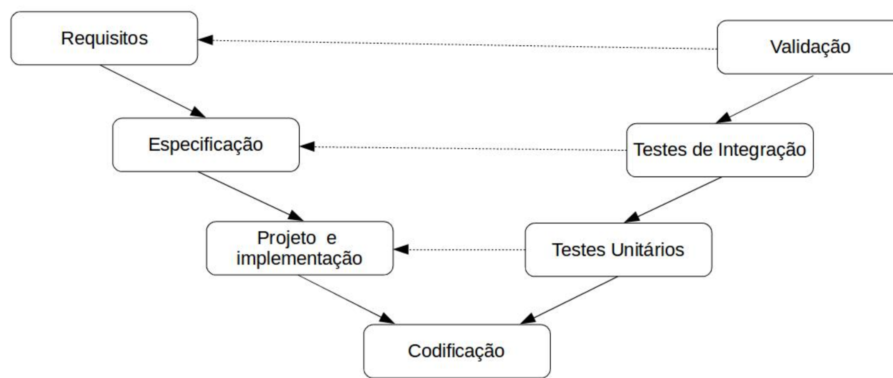


Figura 10 – Representação do Escopo das Fases de desenvolvimento do Modelo Operacional *Multi-Tarefas Qt*.

Um predicado atômico pode ser qualquer coisa/algo que precisamos quantificar. Esse conceito pode ser observado no exemplo da Figura 11, podemos verificar dentro do contexto exposto na ilustração, dois objetos, o objeto *thrd1* e o objeto *thrd2*, podemos dizer que o objeto *thrd1* recebe *num=1*, podemos escrevê-lo na forma "recebe(*thrd1*,num)", podemos fazer o mesmo para o objeto *thrd2* e escrevê-lo na forma, "recebe(*thrd2*,num)", onde *num=2*. Para afirmar que o objeto *thrd2.num* é maior que *thrd1.num*, podemos usar o predicado "maior"e escrevê-lo como "maior(*thrd2.num*",*thrd1.num*).

Podemos expressar os quantificadores das variáveis da Figura 11 da seguinte forma:

$$recebe(thrd1, num) \wedge recebe(thrd2, num) \wedge maior(thrd2.num, thrd1.num) \quad (4.1)$$

De tal forma, podemos dizer que para todo objeto criado é atribuído um valor a *num* , pela seguinte fórmula CTL:

$$(\forall)[objeto(X) \rightarrow \exists Y[recebe(X,Y)]] \quad (4.2)$$

```
1 using namespace std;
2
3 class Mythread : public QThread{
4         public:
5         int num;
6 };
7 int main(int argc, char *argv[])
8 {
9     Mythread thrd1, thrd2;
10    thrd1.num=1;
11    assert(thrd1.num==1);
12    thrd2.num=2;
13    assert(thrd2.num==2);
14
15    assert(thrd1.num > thrd2.num);
16 };
```

Figura 11 – Exemplo de validação da criação do objeto de classe, com o auxílio da inclusão de assertivas.

4.3.2 Explorando a Árvore de Acessibilidade

Podemos demonstrar quais são os estados alcançáveis de um programa multi-tarefas. Em especial, podemos usar uma árvore de alcançabilidade (do inglês *Reachability Tree* (RT)), que pode ser obtida a partir do desdobramento das *threads* que estão em execução em um programa, uma vez que consideramos que qualquer *threads* em execução $j \in T$ pode fazer transições de estados, para tornar possível computar todos os estados que a *thread* poderá alcançar durante o seu processo de execução.

Definição 4.1 *Para um programa multi-tarefas com n threads ativas, cada nó na RT é uma tupla* $v = \left(A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n\right)$ *para um dado intervalo de tempo i, onde:*

- A_i *representa a thread atualmente ativa;*
- C_i *representa o número de trocas de contexto;*
- s_i *representa o estado atual da thread;*
- l_i^j *representa a localização atual da thread j;*
- G_i^j *representa as proteções de fluxo de controle acumulados na thread j ao longo do caminho de* l_0^j *para* l_i^j

Considerando que cada uma das *threads* comunicam-se por intermédio de variáveis globais, consideramos, para fins de validação, apenas as trocas de contexto na RT realizadas em instruções

visíveis, que estão presentes em pontos de sincronização e instruções que contenham variáveis globais, como pode ser observado na Figura 11.

Como em Gupta et al. (GANAI; GUPTA, 2008), não modelamos opções de trocas de contexto dentro de instruções visíveis individuais. Isso é seguro desde que as instruções apenas leiam ou escrevam em uma única variável global, mas em geral isso é uma sub-aproximação. Entretanto, não foi identificado nenhum problema nos nossos *benchmarks* que implementam a concorrência no *Qt*. Além disso, durante a construção do modelo, não modelamos as trocas de contexto entre um teste de fluxo de controle e a próxima instrução visível, pois este teste não pode influenciar no estado do sistema (CORDEIRO, 2011).

A fim de expandir a *RT* e explorar todas as possíveis intercalações, o *ESBMC* executa simbolicamente cada instrução do programa *GOTO* multi-tarefas. Isso leva como entrada o programa e o nó *RT* atual e gera seus filhos de acordo com um conjunto de regras. O *ESBMC* assume ainda que a RT é expandida em um nó v no tempo i e o guarda no fluxo de controle $G_i^{A_i}$ que foi acumulado na *thread* t^{A_i} que está habilitado no estado s_i (ou seja, a fórmula correspondente é satisfatível), de tal forma que, a *thread* possa potencialmente executar a instrução I no local $l_i^{A_i}$ (CORDEIRO, 2011).

```
1
2 struct critical_region{
3    int status;
4    int type_priority;
5    int ID_thread;
6    int running;
7    int pause;
8    int block;
9 };
10
11 struct mutex{
12         int status;
13         critical_region cs;
14         mutex *prox_cs;
15
16 };
```

Figura 12 – Struct com variáveis compartilhadas *CriticalSection*, usadas no gerenciamento dos recursos compartilhados.

No entanto, duas observações importantes podem nos ajudar. Primeiro, a maioria dos *bugs* de concorrência em aplicações reais são superficiais, onde apenas algumas mudanças de contexto são necessárias para expor as falhas (REHOF; QADEER, 2005). Consideramos as *threads*, partindo da hipótese de um *heap* e uma pilha limitada das funções (*stack*) (REHOF; QADEER,

2005), de tal forma que a pilha de execução nada mais é do que uma área que foi reservada dentro do espaço de endereçamento do processo principal, que funciona como uma estrutura de dados tipo LIFO (*Last-In First-Out*). Essa arquitetura é usada como base para a implantação do sistema de prioridades, ou seja, durante a execução quando uma nova *threads* é criada em um bloco de memória local, a mesma é criada e empilhada no topo da pilha. Nesse bloco de memória, há uma referência para todas as variáveis que foram criadas ou que estão sendo apontadas pela pilha de execução e que estão alocadas no *heap* (área reservada para memória global do programa) e uma prioridade é adicionada a elas, o que permite ao agendador de prioridade gerenciar a ordem de execução, de cada uma das *threads* em execução. Neste trabalho utilizamos as técnicas BMC para detectar onde ocorrerão as violações de propriedades de segurança.

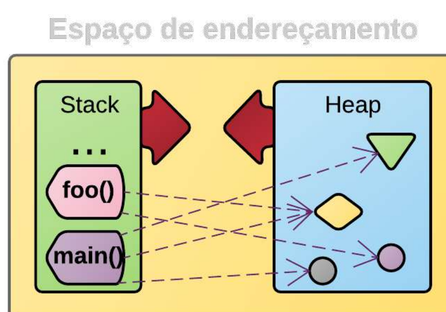


Figura 13 – Representação do Funcionamento do Espaço de Endereçamento no Modelo Operacional *Multi-Tarefas Qt*.

Como pode ser visto na Figura 13, quando o modelo é carregado, automaticamente o sistema operacional disponibiliza um espaço de endereçamento, onde serão armazenados a pilha de função, dados, variáveis e ponteiros necessários, sempre que uma função é chamada, ou desalocada, além da memória *heap*, onde as variáveis e os dados são alocados sempre que uma nova *threads* é criada ou acessada. Uma das principais contribuições desse trabalho é fornece uma abordagem completa da semântica da concorrência encontrada no *framework* Qt, a partir de uma modelagem assíncrona, para a execução das *threads*.

Note que a concorrência dentro do *framework* Qt ocorre de forma assíncrona, onde o gerenciamento ocorre a partir de um sistema de prioridades implementado pelo próprio compilador, que conta com um escalonador com 8 tipos de prioridades distintas, lançadas em tempo de execução, onde cada *thread* entra na pilha de execução a partir da prioridade atribuída à ela.

Tabela 3 – Na tabela a baixo e possível observar os tipos de prioridades que uma thread pode assumir no momento de sua execução.

Constantes	**Valores**	**Descrição**
QThread::IdlePriority	0	A thread é agendada apenas quando nem uma outra thread estiver execuando
QThread::LowestPriority	1	A thread é agendada com menos frequência que LowPriority.
QThread::LowPriority	2	A thread é agendada com menos frequência que NormalPriority.
QThread::NormalPriority	3	A prioridade padrão do sistema operacional
QThread::HighPriority	4	A thread é agendada com mais frequência que NormalPriority.
QThread::HighestPriority	5	A thread é agendada com mais frequência que o HighPriority.
QThread::TimeCriticalPriority	6	A thread é agendada com a maior frequência possível.
QThread::InheritPriority	7	A thread usa a mesma prioridade que a thread pai. Este é o padrão.

No processo de modelagem, damos atenção principal na redução do tamanho das instâncias de problemas do BMC, buscando uma análise mais profunda. Em especial, buscamos diminuir assim o tempo de verificação, uma vez que, como já foi mencionado, quanto maior o número de instruções adicionadas ao código a ser verificado, maior a quantidade de intercalações que são geradas no decorrer da verificação. Além disso, buscamos adicionar as intercalações de forma preguiçosa, onde as restrições são adicionadas conforme é necessária a checagem de determinada propriedade, o que reduz a profundidade da RT que é gerada durante o processo de verificação, o que ajudou também na diminuição da complexidade do processo de busca.

Durante a construção do modelo introduzimos ciclos de espera para modelar as intercalações das *threads*, desenvolvemos também um escalonador implicitamente, sempre tomando cuidado com os ciclos de espera, uma vez que podem ser prejudiciais ao desempenho do *BMC*.

4.3.3 Fluxo de Controle

Para realizar a construção do modelo, considerou-se inicialmente o fluxo principal do ciclo de execução das *threads* dentro do *framework Qt*. Identificamos inicialmente as bibliotecas principais de criação e manipulações das *threads* dentro do *Qt*. Partindo daí, analisamos cada um dos métodos, classes e subclasses, presentes nas bibliotecas analisadas, de forma que o modelo proposto satisfaça as fórmulas lógicas proposicionais, as quais foram representadas pelo uso de assertivas, que dentro do modelo nada mais são do que sentenças abstratas da linguagem alvo da verificação.

Desta forma, podemos realizar a validação das prioridades do sistema alvo da verificação, na forma de fórmulas de Lógica de Tempo Linear (LTL) (BIERE et al., 2006), de modo que as proposições, ao fim da verificação das propriedades que representam o comportamento do sistema, puderam ser verificadas em todos os possíveis estados que o sistema poderia assumir no decorrer da verificação. A checagem destas propriedades retorna verdadeiro ou falso, ao

fim da execução, sendo utilizadas para a verificação de pré-condições e pós-condições para que as estruturas alvos da verificação possam ser verificadas de forma satisfatória. O processo de validação do modelo foi possível a partir da utilização de programas mais simples, visando avaliar o maior número de cenários quanto o possível, buscando diminuir a complexidade do modelo.

Com isso, buscamos garantir que algoritmos eficientes realizem a verificação, se uma determinada fórmula que é responsável pela representação dos estados do sistema $s \in S$, estão sendo atendidos $\models$ corretamente, pelas estruturas de dados que estão sendo verificadas pelo modelo $\widehat{W}$. De forma que a problemática possa ser representado pela expressão:

$$\widehat{W} \models S \tag{4.3}$$

Em geral, as condições de segurança, tais como a ausência de impasses e a ausência de violações em afirmações, enquadram-se nessa categoria e são alvos muito atraentes para verificações exaustivas e automatizadas.

4.3.4 Threads Lançadas em Tempo de Execução e *Link* de Ponteiros

As *threads* foram organizadas na memória como grafos, que utilizam registros de controle, organizados em um fluxo de controle, onde são aplicadas as políticas de controle de agendamento, para que possam ser implementadas as políticas de prioridades presentes no *framework Qt*, uma vez que programas concorrentes são de natureza não determinísticas.

As *threads* são modeladas partindo da ideia de uma árvore computacional, que é modelada a partir de um sistema de transição de estados (do inglês *Labelled Transition System*, LTS) (TRETMANS, 2008), que nada mais é que um modelo, que incluem os estados e as transições de sistema. Esse LTS é modelado em formato de tuplas, onde os objetos associados as *threads* são separados do *offset* (corresponde ao número de locais de endereços adicionados ao endereço base) dentro do objeto.

Além de ponteiros de *heap* regulares, existem 3 tipos de ponteiro adicionais: global, constante e ponteiros. Embora todos os dados (em oposição ao código) estejam armazenados no *heap*, não é o caso de cada variável global ou de cada constante, residir em um objeto *heap* separado. Em vez disso, a máquina virtual usa a alocação baseada em *slot* para esses tipos de dados, ou seja,

um único objeto de *heap* para variáveis globais e outro para dados constantes. (BARNAT et al., 2013)

Um ponteiro global ou constante (diferenciado de ponteiros de *heap* por uma marca de tipo de 2 bits) refere-se a *slots* dentro do objeto do *heap* global designado. Os limites de espaço são impostos como limites de objeto. A distinção entre ponteiros de *heap* e outros tipos de ponteiro é importante quando o sistema operacional deseja implementar a semântica tipo *fork()*: Utilizamos a semântica fork(), no modelo operacional que propomos, nela as threads são modeladas em processos distintos com variáveis globais baseadas em *slots* sempre que uma nova thread entra em execução. Nesse cenário, um novo processo é criado e executado em paralelo com a thread que está em um processo anterior, o novo processo em que a nova thread criada executa, e idêntico ao processo anterior. Em virtude disso, é possível que processos diferentes compartilhem o mesmo código (e constantes). O sistema operacional pode definir um registro de controle para informar à máquina virtual qual objeto *heap* atualmente mantém variáveis globais (ROČKAI et al., 2018). Essa abordagem é possível graças à capacidade dos verificadores de modelos atuais suportarem a indexação de ponteiros.

4.3.5 Detectando Corridas de Dados e Proteção de memória

Mesmo que os verificadores baseados em BMC não reconheçam as *threads* como objetos de primeira classe, é possível acomodar a verificação de programas multi-tarefas, de maneira que, o modelo operacional proposto executa as instruções de forma sequencial, ou seja, é possível reconfigurar o modelo, de tal forma que possa ocorrer um reescalonamento após cada acesso à memória, possibilitando, assim, que as intercalações executadas pelas *threads* que ocorrem no decorrer da execução dos testes possam ser simuladas. Conseguimos, desta forma, detectar erros de corrida de dados, nas suítes de testes utilizadas.

O modelo operacional que propomos compreende um conjunto finito de *threads* determinísticas, comunicando-se por intermédio de variáveis compartilhadas, usadas com o principal objetivo de sincronização de operações de o bloqueio/liberação de recursos, tendo sua execução estritamente controlada.

Para que o processo de sincronização das threads, dentro do Modelo Operacional Simplificado Multi-Tarefas Qt, ocorra de forma controlada, desenvolvemos um Modelo de Estabilidade de Memória (MEM), definimos as propriedades para a criação do MEM (QADEER; REHOF, 2005),

seguindo as regras seguintes:

- Regra de ordem de acesso às variáveis compartilhadas: As *threads* executam de forma individual e o acesso aos recursos compartilhados, tais como, leitura/escrita devem ser realizados de forma individual;
- Regra de uso dos recursos compartilhados: Cada *thread* tem um tempo máximo para a alocação de um recurso compartilhado, com o intuito de evitar problemas de *Lockout* (trancamento), onde várias *threads* ficam aguardando um recurso que encontrasse alocado, ser liberado para outra *thread*. Tal estratégia ajuda a reduzir o tempo de processamento e evita que ocorram inconsistências;
- Regra de leitura/escrita em variáveis compartilhadas: Todas as variáveis compartilhadas devem armazenar as alterações realizadas pelas interações anteriores;
- Regra de acesso a Exclusão Mútua: O acesso compartilhado às operações de bloqueio e desbloqueio, devem ocorrer por intermédio de mecanismos modelados partindo da utilização da técnica de Exclusão Mútua (mutex), por meio das operações de *lock* (bloqueio) e *unlock* (desbloqueio);

Implementamos um mecanismo de Exclusão Mútua (EM), para impor uma separação do espaço de endereçamento onde as *threads* são executadas: Se uma thread em particular não possui acesso a um ponteiro para a EM, esse objeto não pode ser acessado.

A única maneira de acessar a EM é obter um ponteiro para um objeto associado a EM do tipo mutex, esse acesso pode ser ser facilmente evitado pelo sistema operacional com a utilização da técnica de utilização de mutex. A única armadilha dessa abordagem é a implementação da Comunicação entre Processos (ICP). Ou seja, a imposição da proteção da memória depende da capacidade do EM de invalidar o acesso de ponteiros solicitados por threads via ICP. O sistema operacional deve preservar os ponteiros enviados por meio do ICP para outro processo e, em seguida, retornar da mesma maneira, ao mesmo tempo em que aplica o isolamento do processo, o que fornecer um mecanismo de conversão. Advertências similares se aplicam a segmentos de memória compartilhada que podem conter ponteiros para eles mesmos ou para outros segmentos (BARNAT et al., 2013).

A comunicação que ocorre entre as *threads*, situadas em processos distintos, foi modelada por intermédio do uso e da manipulação de *threads* de usuário, usando chamadas de sistema, pois

não existe a possibilidade de acesso às variáveis comuns a ambos. No entanto, essa abordagem pode ser ineficiente caso a comunicação seja muito volumosa e frequente, ou envolve muitos processos. Por essas razões, modelamos uma área de memória comum para que as *threads* possam ser acessadas de forma direta e fácil, pelos processos interessados, sem o custo da intermediação do núcleo do Sistema Operacional (SO) (MAZIERO, 2014). A qual chamamos de Área de Exclusão Mútua.

4.4 Concorrência e Acesso à Memória Compartilhada

Os verificadores de modelos atuais não reconhecem todas as estruturas do *framework* Qt, a partir do modelo que propomos eles devem ser capazes de identificar e verificar estruturas de dados que implementem programas multi-tarefas suportados pela plataforma Qt. O modelo operacional foi desenvolvido utilizando a linguagem de programação C/C++ e faz uso da biblioteca Pthread/POSIX (Portable Operating System Interface, 2015).

As simplificações do fluxo de controle desses requisitos são descritos nas seções seguintes. Como tal verificação afeta o acesso à memória, o nosso modelo operacional (MO) foi projetado para executar as instruções de forma sequencial, de tal forma que, ao executar o MO, as operações de memória são executadas, permitindo, assim, que ocorra um reescalonamento, sempre que necessário, após cada acesso a memória. Este procedimento de acesso à memória será mais bem explorado nas próximas seções.

4.4.1 Memória escalonável

Nos verificadores de modelo tradicionais, as *threads* são gerenciadas pelos verificadores como objetos de primeira classe (ROČKAI et al., 2018), isto é, uma entidade que pode ser criada, destruída, passada para uma outra função é retornada como um valor de forma dinâmica. Com isso, as *threads* são implementadas e compiladas dentro do SO de forma virtual, usando interfaces de chamadas ao ID da thread principal, o que simplifica o modelo, transferindo a responsabilidade do gerenciamento, para o verificador de modelos que estiver sendo utilizado em conjunto com o modelo operacional proposto. Modelamos as chamadas de criação, manipulação e delimitação dentro da Área de Seção Crítica, mediante a utilização da biblioteca de manipulação de *threads* e processos *pthread* nativa da linguagem de programação C, utilizando o padrão

POSIX (Portable Operating System Interface, 2015), o que tornou a implementação mais fácil e a correção de erros menos crítica.

O *framework Qt* possui um sistema de prioridades próprio, com diferentes níveis de prioridades, como pode ser observado na Figura 4. Para que pudéssemos validá-lo no modelo que propomos, foi necessário a criação de um Escalonador de Prioridades (EP), que é o responsável por prover o agendamento que decide qual *thread* deve ser executada e em que ordem as *threads* irão executar, foram ainda implementadas rotinas de interrupções de sistema para retornar o controle ao EP. Sempre que uma nova *thread* é executada, primeiramente é observado se o nível de prioridade da *thread* que entra em execução é maior que a prioridade das demais. Neste caso, o sistema de interrupções é acionado, para que a *thread* de maior prioridade tenha acesso ao recurso alocado.

É importante ressaltar que as rotinas que envolvem o acesso as variáveis locais do agendamento ou qualquer função que as chama, não são armazenadas. No entanto, os estados do agendador e um ponteiro para o ID da *thread* devem ser armazenado no *heap*, de modo que a partir desses dados registrados, o agendador possa acessar/modificar os dados armazenados no *heap*. De tal forma, o agendador tem as rotinas de agendamento das prioridades que o escalonador deve assumir durante a execução das verificações.

Foram modelados 07 tipos de transições de espaços de estados em que uma determinada *thread*, poderá se encontrar, *start()*, *run()*, *exec()*, *QeventLoop()*, *quit()*, *exit()* e *quit()*, que correspondem ao ciclo inicial de criação e manipulação das *threads* no Qt, como pode ser visto na Figura 14.

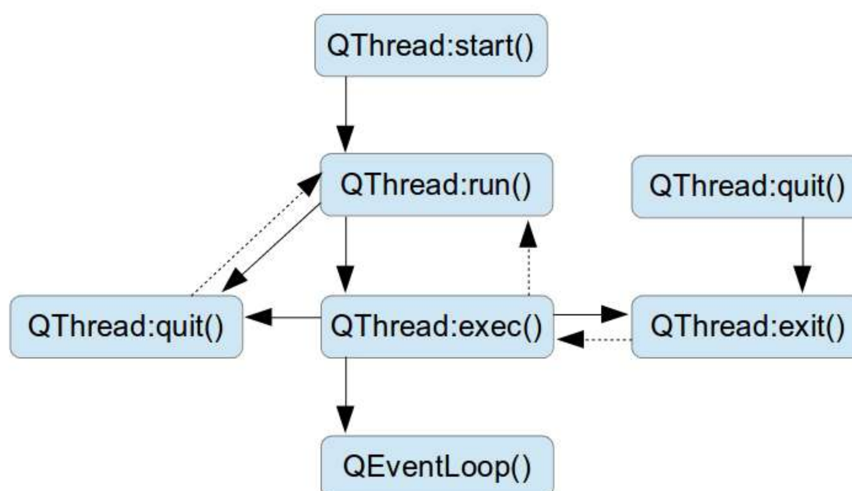


Figura 14 – Principais estados que a *thread* pode assumir em tempo de execução.

De tal modo que, quando uma *thread* necessita obter o acesso a um estado específico, o escalonador aciona o Agendador de Prioridade (AP), que decide qual *thread* irá executar por meio de escolhas não-determinísticas e, dependendo da análise do AP, o recurso é transferido para a *thread*. Nesse ponto é criado um ponteiro que guarda o estado da thread no AP.

A partir da junção de verificadores BMC e do MO proposto, os espaços de estados gerados no decorrer da verificação podem ser sistematicamente explorados. Em especial, o verificador de modelos realizará a análise do código fonte, onde os estados atuais e os estados sucessores serão modelados por meio de um grafo a ser verificado. O AP, que integra o escalonador, não é quem gerencia as ramificações que os espaços de estados assumiram durante o processo de verificação, mas sim os verificadores BMCs.

Durante o desenvolvimento do MO não tratamos explicitamente as seções atômicas, uma vez que os verificadores de modelos BMC atuais já possuem suporte a instruções de acesso à memória, de forma que foi possível simular a Área de Exclusão Mútua. Entretanto, é importante ressaltar que isso não é o suficiente, visto que é impossível recuperar a relação entre operações atômicas ou bloqueios explícitos de acesso à memória compartilhada que são protegidos, sendo essa uma das principais razões pelas quais precisamos de verificadores de modelos (ROČKAI et al., 2018).

4.5 Modelando Primitivas de Sincronização no *QThread*

Nesta Seção, apresentamos nossa abordagem para a modelagem das primitivas de sincronização da biblioteca *Qthreads* (SOUZA; CORDEIRO; JANUARIO, 2018). Partindo das premissas elencadas na Seção 4.3.1, assumimos que o modelo operacional simplificado multi-tarefas Qt está em conformidade com as especificações da biblioteca *Qthread*. Com isso, concentramos todos os nosso esforços na verificação apenas das funções que as utilizam, tendo como intuito validar as estruturas implementadas no modelo.

Os verificadores de modelos atuais em sua grande maioria fornecem suporte à verificação de programas multi-tarefas utilizando a biblioteca *pthread* nativa do C, chamada POSIX (Portable Operating System Interface, 2015). Por essa razão, durante a construção do modelo, tal biblioteca foi utilizada para modelar a biblioteca *QThread*, com o intuito de tornar possível a simulação da criação e a manipulação das *threads* pelo modelo operacional proposto. Mostramos, em nossos experimentos, que nossa modelagem é capaz de detectar o uso incorreto das funções e também é

capaz de detectar operações de bloqueio que podem levar a *deadlocks* locais e globais, como pode ser visto na Seção 5.4.3.

4.5.1 Modelando Operações de Bloqueio de *QThread*

A biblioteca *Pthread* suporta duas funções que utilizamos para implementar a Área de Exclusão Mútua. A Área de Exclusão Mútua nada mais é do que a parte do código modelada para que possam ser realizados os acessos à memória ou aos recursos compartilhados com o intuito de evitar condições de corrida, onde duas ou mais *threads* tentam obter acesso aos mesmos recursos de forma simultânea (REHOF; QADEER, 2005).

Em especial, para a delimitação da Área de Exclusão Mútua, utilizamos as funções *pthread_mutex_init(&mutex)* e *pthread_attr_destroy(&mutex)*. Ambas as funções aceitam como argumento de entrada uma estrutura de dados chamada $mutex(MUTualEXclusion)$, que nada mais é do que um *lock* que pode estar em posse de apenas uma única *thread* por vez, o que garante a exclusão mútua, de forma que as outras *threads* que solicitarem acesso a esse *lock* ficam bloqueadas até que o *lock* esteja liberado. Para obter o lock do *mutex*, utilizamos a função $pthread_mutex_lock$, bloqueando a função caso esta não esteja bloqueada, travando assim o *mutex*. Para desbloquear o *mutex*, utilizamos a função $pthread_mutex_unlock$. Caminhos de computação são bloqueados em um *mutex* quando uma *thread* tenta bloquear um *mutex* que já tenha sido bloqueado por outra *thread* (CORDEIRO, 2011).

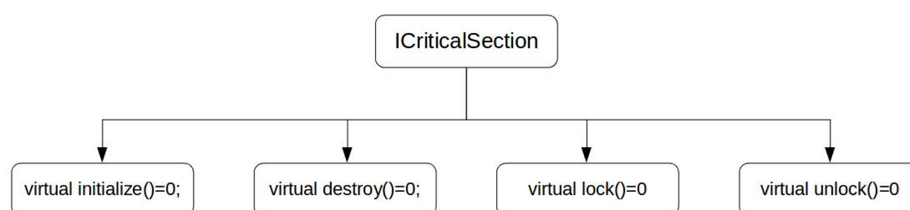


Figura 15 – Escopo principal da Seção Crítica.

Para evitar o problema das condições de corrida, na construção do modelo operacional como pode ser observado nas Figuras 16 e 15, implementamos uma Área de Exclusão Mútua, definindo regiões críticas onde os processos podem ser executados, de tal forma que, quando um recurso estiver alocado em uma região crítica, nenhuma outra *thread* pode ter acesso a esse recursos que foram alocados nela, utilizando a técnica de Exclusão Mútua. Tal técnica nos garantiu obter propriedades:

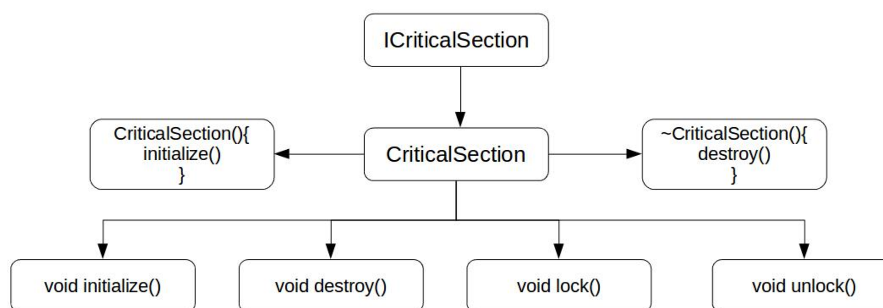


Figura 16 – Escopo da Seção Crítica.

1. Atomicidade: uma vez que ocorre o travamento do *mutex*, conseguimos garantir que ocorram operações atômicas por intermédio da utilização da biblioteca *Pthread*, a qual nos permite alocar um *mutex* para a *thread* que tenha a prioridade no processo de execução. Visto que o mutex esteja travado, nem uma outra *thread* terá acesso aos recursos que estejam alocados dentro do *mutex*, que se encontra bloqueado, até que o *mutex* seja destravado e o recurso liberado para ser alocado a outra *thread*.

2. Singularidade: se o *mutex* estiver bloqueado por uma *thread*, não poderá ser alocado por outra *thread*, até que a *thread* que o bloqueou o libere para ser utilizado por outra.

3. Sem espera ocupada: caso uma *thread* solicite acesso a um *mutex* travado, a *thread* fica em um estado de suspensão até que o *mutex* seja liberado. Quando o *mutex* é liberado, se a *thread* for a próxima da fila de prioridade, é permitido que o *mutex* seja alocado para essa *thread*.

4.5.2 Agendamento de Prioridades no Qt

No Qt usa-se como padrão para a implementação e o gerenciamento de *threads* a biblioteca de sistemas *Qthread* (The Qt Company Ltd. Documentation, 2015). Esta biblioteca fornece os mecanismos necessários para o gerenciamento das *threads* na plataforma, com o auxílio de sinalizadores, sempre que uma *thread* é *started()* e *finished()*, ou quando as funções *isFinished()* e *isRunning()* podem ser usadas.

Como já mencionado anteriormente Seção 4.4.1, a *thread* por padrão começa a execução pelo método *start()*. Desta forma, o sistema operacional agenda a prioridade do *thread* de acordo com

```
1 class ICriticalSection {
2 public:
3     virtual void initialize()=0;
4     virtual void destroy()=0;
5     virtual void lock()=0;
6     virtual void unlock()=0;
7
8
9 };
10 class CriticalSection : public ICriticalSection {
11 public:
12         pthread_mutex_t mutex;
13         CriticalSection()
14         {
15             initialize();
16         }
17         ~CriticalSection() {
18             destroy();
19         }
20         void initialize()
21         {
22             pthread_mutex_init(&mutex, NULL);
23             ...
24         }
25         void destroy()
26         {
27             pthread_mutex_destroy(&mutex);
28             ...
29         }
30         void lock()
31         {
32             pthread_mutex_lock(&mutex);
33             ...
34         }
35         void unlock()
36         {
37             pthread_mutex_unlock(&mutex);
38         ...
39         }
40     };
```

Figura 17 – Assinatura das funções da Área de Exclusão Mútua

o parâmetro de prioridade que é atribuído no momento da chamada da função *start()*, mostrado na Figura 18.

```
1 void QThread :: start ( QThread ::
      Prioridade prioridade =
      InheritPriority)
```

Figura 18 – Assinatura do método *start()*, no *framework* Qt.

O Qt fornece 08 tipos de prioridades diferentes do tipo *enum*, que podem ser vistos na

Tabela 4:

```
1 enum Priority {
2   IdlePriority=0,
3   LowestPriority=1,
4   LowPriority=2,
5   NormalPriority=3,
6   HighPriority=4,
7   HighestPriority=5,
8   TimeCriticalPriority=6,
9   InheritPriority=7
10 };
11   void setPriority(Priority priority);
12   //setPriority --sets the priority of
         a running thread
13   Priority priority() const;
14   void start(Priority =
         InheritPriority);
```

Figura 19 – Assinatura do enumerador *Priority*, no *framework* Qt.

```
1 void QThread::start(Priority priority)
      {
2   if (priority == InheritPriority){
3     assert (ret == 0);
4     if((pthread_create(&_id, &_attr,
          function, (void *)this) )!= 0)
           {
5       exit(EXIT_FAILURE);}
6       pthread_exit(NULL);
7       pthread_detach(_id);
8     } else { setPriority(priority); }
9 };
```

Figura 20 – Representação do método *start()*, no modelo operacional proposto.

Implementamos as políticas de agendamento de prioridades utilizando o método *thread_attr_ setschedpolicy*, da biblioteca *Pthread*, o que nos permitiu definir as diretivas e os parâmetros do escalonador de prioridades.

4.5.3 Modelando Operações de Bloqueio e Exclusão Mútua

No processo de desenvolvimento do modelo operacional, utilizamos a técnica de exclusão mútua, que possibilita assegurar o acesso exclusivo a um recurso que é compartilhado entre as *threads* do programa que está em execução, o que possibilitou reproduzir corridas de dados na Região Crítica. Tomamos como principais requisitos para a implementação da área de exclusão

```
1  Mythread::Mythread() {
2    int num = 0;
3    int g = 0;
4  }
5  void Mythread::run(){
6    int a;
7    a=*((int *)arg);
8    assert(a==10);
9    g=1;
10 }
11 int main(int argc, char *argv[])
12 {
13   // QCoreApplication a(argc, argv);
14   Mythread thr1, thr2 ;
15   thr1.num= 10;
16   thr2.num=1;
17   thr1.start(QThread::HighestPriority)
       ;
18   thr2.start(QThread::HighestPriority)
       ;
19   assert(g = 0);
20   //return a.exec();
21 }
```

Figura 21 – Exemplo da implementação do método *start* com atribuição de prioridade.

mútua os mesmos descritos na Seção 4.3.5. Em especial, a biblioteca *Pthread* suporta duas funções que possibilitaram a implementação da Área de Exclusão Mútua, entre as *threads* chamadas de *pthread_mutex_lock e pthread_mutex_unlock*.

4.5.4 Modelando as Condições de Corrida é Sinalizadores

Segundo Codeiro (CORDEIRO, 2011), durante o processo de verificação de programas multi-tarefas é importante verificar quando ocorrem condições de corrida e violações de atomicidade, que constantemente são apontadas como as principais fontes de problemas de erros em programas concorrentes. Em geral, os problemas podem ocorrer durante as trocas de contexto entre as *threads*, o que é difícil de reproduzir e depurar.

As condições de corrida ocorrem normalmente quando duas (ou mais) *threads*, solicitam o acesso a um recurso simultaneamente e pelo menos uma das *threads* realiza a operação de leitura e gravação. Por exemplo, podemos assumir que duas threads *thr1* e *thr2* desejam incrementar o valor de uma variável global *int i*. De imediato, temos a seguinte sequência de operações:

Na Figura 22, os construtores e às variáveis globais e locais são instanciadas e carregadas na memória, em seguida as *threads* $thr1$ e $thr2$ são inicializadas pelo método $start()$. O $thr1$ lê

```
1 Qthread::Qthread() {
2   int i = 0;
3 }
4 void Qthread::thr1() {
5   int y_{1} = 0;
6   y_{1}= i;
7   y_{1} = i+1;
8   i = y_{1}
9   assert(i==1);
10 }
11 void Qthread::thr2() {
12   int y_{2} = 0;
13   y_{2}= i;
14   y_{2} = i+1;
15   i = y_{2}
16   assert(i==2);
17 }
18
19 int main(int argc, char *argv[]) {
20   Qthread thr1, thr2 ;
21   thr1.start();
22   thr1.exc1();
23   thr2.start();
24   thr2.exc2();
25   assert(g = 2);
26 }
```

Figura 22 – Exemplo de ocorrência de condição de corrida com violação de atomicidade.

o valor da variável i (que é igual a 0) e armazena na variável y_1, em seguida ela incrementa o valor de y_1. Em paralelo, *thr*2 também é inicializada e armazena na variável y_2 o valor de i (que é igual a 0), em seguida *thr*2 incrementa o valor de y_2. Em seguida, y_1 armazena o seu valor (y_1 = 1) na variável i, em seguida y_2 também armazena o seu valor (y_2 = 1) na variável i, quando o *assert* na linha 16 é executado, resultando em um erro, já que o resultado de $i = 1$, quando na verdade, deveria ser $i = 2$. Isso ocorre porque as operações de incremento não são atômicas, ou seja, as operações das *threads* não são interrompidas enquanto ocorre o acesso a um recurso comum entre as duas (no nosso caso a área de memória de i). Isso gera as chamadas condições de corrida, levando a erros de atomicidade. O erro de atomicidade pode ser resolvido por intermédio da criação de operações atômicas, que ocorrem quando uma determinada operação deve ser executada completamente em caso de sucesso, ou abortada caso ocorra algum erro no decorrer da execução.

A forma que utilizamos para modelar e aplicar a atomicidade entre as intercalações que as *threads* realizam durante a execução do programa, foi por intermédio de blocos atômicos (sequências de instruções cuja a execução não é interrompida por outras *threads*). Quebramos

as declarações visíveis em dois blocos atômicos, caso haja mais de uma variável global sendo acessada. O primeiro bloco armazena as variáveis temporais e o segundo verifica se as variáveis temporais atendem às condições. Como pode ser observado no exemplo da Figura 23, modelamos as seções atômicas com os comandos *__VERIFIER_atomic_begin*(), *__VERIFIER _ atomic _ end*(), que usamos como delimitadores dos blocos atômicos, indicando que o bloco não pode sofrer preempção por outra *thread* (CORDEIRO, 2011).

```
1 Qthread::Qthread() {
2   int x =1 , y = 3 , aux1, aux2, soma;
3   void __VERIFIER_atomic_begin();
4   void __VERIFIER_atomic_end();
5 }
6 Qthread::~Qthread(){}
7
8 void Qthread::thr1() {
9   // this is atomic!
10  __VERIFIER_atomic_begin();
11  aux1 = x;
12  aux2 = y;
13  __VERIFIER_atomic_end();
14 }
15 void Qthread::thr2() {
16  // this is atomic!
17  __VERIFIER_atomic_begin();
18  assert(aux1==x && aux2==y);
19  soma= x+y;
20  __VERIFIER_atomic_end();
21 }
22 int main() {
23  QThread thr1;
24  thr1.start();
25  assert(soma = 4);
26 }
```

Figura 23 – Exemplo de modelagem de atomicidade em declarações visíveis.

4.5.5 Resumo

Neste capítulo apresentamos as implementações desenvolvidas para a construção do Modelo Operacional Simplificado Multi-Tarefas Qt/C++, assim como as metodologias para melhorar o desempenho da metodologia proposta o que viabilizou seu uso.

A utilização de um modelo operacional simplificado nos permitiu desenvolver representações das classes, métodos e tipos de dados presentes no *framework* Qt, o que nos permitiu viabilizar a verificação de violações de prioridades inerentes da linguagem adotada. Tal verificação

tornou-se viável pela integração do modelo operacional simplificado multi-tarefas Qt/C++ com o verificador de modelos limitado ESBMC. Para garantir que as propriedades inerentes aos módulos que foram desenvolvidos estivessem sendo verificadas corretamente, baseamos-nos na abstração de predicados, para a modelagem das pré e pós-condições que foram inseridas no modelo operacional, para garantir que as propriedades não estavam sendo violadas, o que aumentou a detecção de falhas no processo de verificação dos *benchmarks* usados durante a validação do modelo.

As *threads* foram modeladas a partir da utilização da biblioteca de sistema Pthread. Em virtude disso, foi possível a reprodução do comportamento dos módulos alvos da verificação, o que tornou possível a criação e a manipulação das threads de sistema. A partir dessa modelagem, foi possível integrar o modelo operacional desenvolvido a verificadores de modelos que utilizem a técnica de verificação de modelos BMC e que fornecem suporte a programas multi-tarefas escritos em C++. A adição do modelo auxilia no processo de verificação dessas estruturas do Qt pelos verificadores BMC usados, sem mudar o comportamento do sistema verificado, ou seja, apenas adicionamos novas funcionalidades a esses verificadores BMC.

5 Avaliação Experimental

Neste capítulo, descrevemos todas as configurações experimentais utilizadas e todos os parâmetros utilizados nas avaliações experimentais, bem como a análise da eficácia e desempenho do modelo proposto, mediante ao uso dos solucionadores SMT (Z3 (MOURA; BJØRNER, 2011), Boolector (BRUMMAYER; BIERE, 2009) e Yices (DUTERTRE, 2014)). Realizamos a análise de todas as suítes de teste desenvolvidas por intermédio de programas multi-tarefas Qt, que foram utilizados durante o processo de validação do modelo que propomos, tendo como base a documentação do Qt (The Qt Company Ltd., 2015). Descrevemos também os resultados que foram obtidos a partir da verificação de todos os casos de testes, presentes nas suítes de testes para a validação da biblioteca *Qthread*.

A validação das suítes de teste desenvolvidas ocorreu a partir da utilização dos verificadores de modelos (ESBMC++ (CORDEIRO et al., 2012), LLBMC (MERZ; FALKE; SINZ, 2012) e DIVINE (BARNAT et al., 2013)), com o intuito de avaliar a eficácia do modelo que propomos em conjunto com outras abordagens de verificação. E, por fim, analisamos os resultados inerentes da comparação entre o comportamento dos módulos *Qthreads* do *framework* Qt, em relação ao modelo proposto com o objetivo de avaliar a conformidade da modelagem proposta com os módulos analisados do *framework* Qt.

5.1 Configuração dos Experimentos

Com o objetivo de validar a eficiência da metodologia proposta, para a verificação de programas multi-tarefas Qt, desenvolvemos um conjunto de suítes de testes automatizadas que denominamos de Modelo Operacional Simplificado Multi-tarefas Qt. Este conjunto de suítes de testes contém 120 *benchmarks*, para a validação de cada um dos métodos, tipos de dados e quantificadores, os quais foram utilizados para a garantia do funcionamento correto do ciclo de vida das *threads* no referido *framework*, tomando como base a documentação do *framework* Qt (The Qt Company Ltd., 2015). Os *benchmarks* foram divididos em 08 principais módulos de testes denominados, *QThread::start()*, *QThread::run()*, *QThread::exec()*, *QEventeLoop()*, *QThread::quit()*, *QThread::exit()*, *QThread::terminit()* e *Qthread::Priorit()*.

É importante ressaltar que todos os *benchmarks* desenvolvidos possuem casos de teste para a

validação de suas proposições de acordo com o método do módulo *Qthreads* que será analisado, que, em sua maioria, possuem acesso aos módulos Qt *Core* e Qt GUI. Os *benchmarks* foram desenvolvidos a partir da documentação referente ao *framework* Qt (The Qt Company Ltd., 2015), como o objetivo de analisar e prestar suporte a todas as característica fornecidas pelo *framework* Qt.

Todas as propriedades dos métodos e funções foram extraídas no formato de pré-condições e pós-condições o que nos permitiu realizar uma análise modular no processo de verificação dos sistemas multi-tarefas (foram validados por intermédio de testes manuais antes de serem adicionados a suas respectivas suítes de testes).

A modularidade, como chave para dimensionar a análises de programas para grandes sistemas de software multi-tarefa, foi usada de forma eficiente em outro trabalho que foi desenvolvido por Freund e Qadeer et al. (FREUND; QADEER, 2004), entretanto ela foi utilizada para verificar programas multi-tarefas inscritos em JAVA, como pode ser visto na Seção 3.6.

As pré-condições adicionadas no modelo indicam quais bloqueios devem ser mantidos após a entrada no método e as declarações para cada objeto protegido que indica o bloqueio que deve ser mantido ao acessar esse objeto.

Desta forma, foi possível estabelecer se o modelo estava apto a identificar se os *benchmarks* usados para a validação do modelo possuem ou não algum erro, isto é, se estão em conformidade com a documentação. As suítes de testes foram classificadas pelo tipo de teste ao qual estão associadas e variam de acordo com os requisitos a serem avaliados. A heurística que estabelecemos para estabelecer os tipos de testes realizados foram duas:

1. Primeiro: Criamos suítes de testes positivas com o intuito de demonstrar que as propriedades foram satisfeitas.

2. Segundo: Criamos suítes de testes negativas, com o intuito de demonstrar que as propriedades são atendidas apenas sobre algumas condições desejadas (ou seja, casos de testes que correspondam a condições ou dados que violem as propriedades que estão sendo validadas).

As suítes de testes foram desenvolvidas através da utilização da técnica de modularização para a construção dos módulos da biblioteca *QThread* o que nos permitiu dimensionar a análise de sistemas multi-tarefas de forma precisa.

A técnica de modularização foi aplicada no processo de construção de cada uma das suítes de testes desenvolvidas, de forma que cada método, função e classes associadas pudessem ser bem definidas e testadas de forma independente. Com isso, buscamos que os módulos desenvolvidos não fossem grandes demais, evitando que os módulos se tornassem multifuncionais e de difícil compreensão, de tal forma as suítes de testes desenvolvidas verificasse apenas um módulo por vez, de tal forma que o módulo analisado apresentasse apenas as estruturas necessárias para atingir o objetivo da verificação das estruturas de dados associadas ao módulo validado.

É importante ressaltar que devido às interações entre as threads, as operações de pré e pós-condições, não é o suficiente para garantir a verificação modular de programas multi-tarefas. Para que a modularização do sistema fosse alcançada, como foi proposto por Jones et al. (JONES, 1983), foi necessário adicionar verificações de propriedades(regras de provas de teoremas) para que fosse possível detectar e verificar as interações entre as *threads*, a fim de garantir que as propriedades inerentes da linguagem alvo das verificações estavam sendo corretamente atendidas.

Com isso, podemos identificar Especificações de Garantia propostas inicialmente por Jones (JONES, 1983), para verificar se todas as propriedade estavam sendo corretamente identificadas, uma vez que as propriedades são extraídas a partir de documentos de especificações da linguagem alvo, em formato de suposições matemáticas, que especificam as suposições sobre o comportamento do módulo que está sendo analisado, buscando, assim, garantir a validação das propriedades garantidas pelo sistema.

Partindo daí, podemos garantir que 59 dos 120 casos de teste negativos (isto é, 49% dos *benchmarks* gerados contêm violações de propriedades. Ex: Erros de atomicidade, deadlock, falhas de segmentações e etc) e 61 casos de teste positivos (isto é, 51% dos *benchmarks* analisados são considerados corretos).

Na realidade, esse tipo de inspeção sobre os *benchmarks* é essencial para a avaliação experimental, comparando-se os resultados que foram obtidos por intermédio dos verificadores de modelos, como conseguinte, avaliar de forma confiável se são erros reais e não falsos positivos foram encontrados.

Todos os nossos experimentos foram realizados em um computador Intel Core i7-4790 com 3.60 GHz de *clock* e 24 GB (22 GB de memória RAM e 2 GB de memória virtual), utilizando um sistema operacional *open source* de 64 bits denominado Ubuntu 16.04. Além disso, durante a avaliação experimental utilizamos o verificador de modelos ESBMC++ v1.25.4 com três solucionadores SMT instalados (Z3 v4.0, Boolector v2.0.1 e Yices 2 v4.1).

Os limites de tempo e memória utilizados para cada *benchmarks* foram definidos em 600 segundos e 22 GB, respectivamente. Por último, mas não menos importante, combinamos o Modelo Operacional Simplificado Multi-Tarefas Qt com os verificadores de modelos LLBMC v2013.1 e DIVINE v3.3.2, com o objetivo de proporcionar comparações entre diferentes ferramentas, tendo em vista que os verificadores implementam abordagens diferentes em seus processos de verificações.

Os períodos de tempo medidos durante o processo de verificação com os verificadores (ESBMC++, DIVINE e LLBMC) em conjunto com o modelo operacional multi-tarefa Qt foram medidos usando uma função da biblioteca *time.h* do módulo *Time*::*HiRes*, com a finalidade de medir os tempos das verificações de cada um dos *benchmarks*, o tempo inicial t_0 e tempo final t_1 das verificações de cada uma das suítes de testes são configuradas pelo parâmetro *clock_gettime* [1], que retorna a hora do sistema operacional no momento da execução do programa. Com isso, para aferir o tempo de verificação para cada uma das suítes, a função *clock_gettime* é chamada no início e no fim de cada verificação, onde o tempo total de verificação é definido pela subtração de t_0 e t_1.

Utilizamos essa função com a finalidade de determinar o período inicial e final da execução dos *benchmarks*. Os períodos de tempo podem ser impressos em minutos (min) ou em segundos (s).

5.2 Uma Comparação Entre os Solucionadores SMT (Z3, Boolector e Yices 2)

É conhecido que diferentes solucionadores SMT podem afetar fortemente os resultados obtidos (BARRETT et al., 2010), tendo em vista que não existe homogeneidade entre a implementação dos mesmos e as heurísticas matemáticas usadas para implementá-los.

Inicialmente, as verificações foram realizadas utilizando o verificador de modelos ESBMC três solucionadores SMT são utilizados no seu processo de verificação (Yices 2, Boolector e Z3). Objetivando a avaliação do desempenho e a eficácia da metodologia proposta, executamos o verificador ESBMC++ v1.25.4 com cada um dos solucionadores suportados individualmente. Considerando este pressuposto, o solucionador que obteve o pior desempenho dentre os três

[1] clock_gettime (CLOCK_REALTIME)

solucionadores foi o Yices, o qual obteve uma taxa de verificações bem sucedidas de aproximadamente 65% das suítes de testes analisadas. O solucionador Yices não conseguiu resolver corretamente as fórmulas SMT que foram geradas pelo processo de verificação do ESBMC em alguns casos específicos. O principal fator que ocasionou o mal desempenho no processo de verificação com o Yeces 2 é decorrente da indisponibilidade do solucionador de prestar suporte a tuplas, lembrando que as threads dentro do processo de compilação do ESBMC são modeladas como tuplas, como pode ser visto na Subseção 2.2.2. Além disso, como pode ser observado na Figura 24, o Yices 2 obteve um tempo de verificação de aproximadamente 25 minutos. Já os solucionadores Boolector e Z3 apresentaram uma taxa de verificações bem sucedidas de 89%. Entretanto, o tempo de verificação dos solucionadores é discrepantes. O solucionador Z3 apresentou um tempo de verificação de aproximadamente 10 minutos, enquanto o solucionador Boolector apresentou um tempo de verificação de aproximadamente 5 minutos, para executar os 120 *benchmarks* analisados.

Partindo dos resultados obtidos, das execuções dos *benchmarks* com o uso solucionadores, observou-se que o Boolector mostrou-se mais rápido e eficaz do que os outros solucionadores, mesmo o solucionador Z3 tendo alcançado a mesma taxa de verificações bem sucedidas que o Boolector. O tempo de verificação no decorrer das verificações com o Z3 é muito elevado quando comparado ao tempo de verificação utilizando o Boolector, considerando, ainda, que o Yices 2 não conseguiu realizar uma análise completa do conjunto de benchmarks analisados. Partindo desse princípio, durante o processo de verificação com o ESBMC, utilizamos o solucionador Boolector, no processo de verificação dos *benchmarks* analisados.

Na Figura 25, é possível analisar o resultado das taxas de cobertura das verificações bem sucedidas de cada um dos solucionadores usados pelo verificador ESBMC, bem como o seu tempo de verificação.

5.3 Avaliação dos Resultados Para o Modelo Operacional Simplificado Multi-Tarefas

Todas as suítes de teste que foram desenvolvidas para a validação do Modelo Operacional Simplificado Multi-Tarefas Qt, as quais foram integradas e verificadas de forma automática pelos

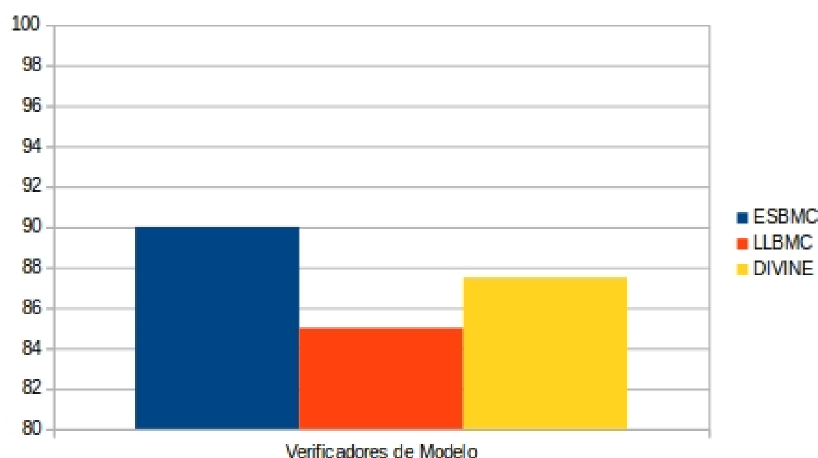


Figura 24 – Resultados da análise dos Tempos de Verificação do ESBMC utilizando os solucionadores Boolector, Z3 e Yices 2 executados em conjunto com o verificador de modelos ESBMC.

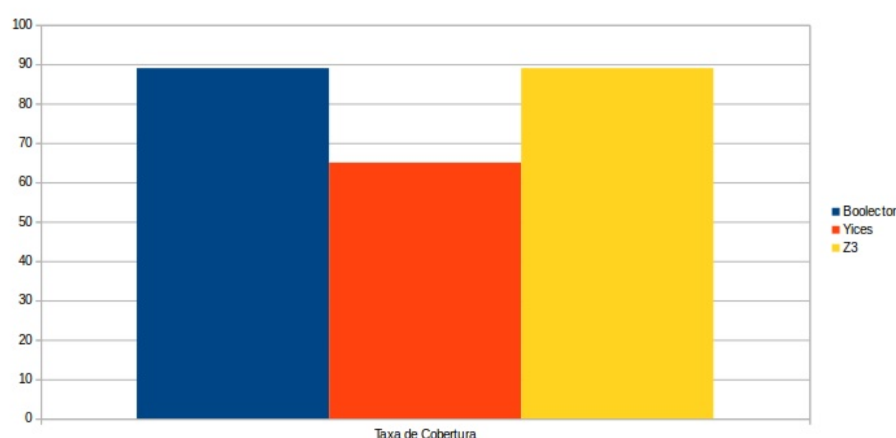


Figura 25 – Resultados da análise da Taxa de Cobertura das verificações do ESBMC utilizando os solucionadores Boolector, Z3 e Yices 2 executados em conjunto com o verificador de modelos ESBMC.

verificadores de modelos ESBMC++, LLBMC (FALKE; MERZ; SINZ, 2013) e DIVINE (BARNAT et al., 2013), com o objetivo de validar a corretude da metodologia proposta.

Por intermédio do ESBMC foi possível realizar a análise e a comparação do comportamento da modelagem proposta com diferentes solucionadores SMT, como descrito na seção anterior 5.2. Podemos também analisar e avaliar a metodologia proposta com outras abordagens de verificação formal de modelos, por intermédio do uso dos verificadores de modelos LLBMC e DIVINE.

É importante frisar que até o presente momento não existe nem uma outra ferramenta de verificação que seja capaz de realizar a verificação de programas concorrentes escritos no *framework* Qt ou mesmo um modelo operacional que se assemelhe com a modelagem do modelo

operacional proposto e que tenha sido desenvolvido em Qt/C++ para a verificação e validação de tais estruturas.

Como o Modelo Operacional Simplificado Multi-Tarefas Qt foi desenvolvido totalmente em C++, ele mostrou-se extremamente versátil, uma vez que essa abordagem de implementação possibilita que o modelo seja inserido no processo de verificação de outros verificadores de modelos que reconheçam e verifiquem estruturas de dados C/C++ e que ofereçam suporte à biblioteca de manipulação de *threads* padrão POSIX (Portable Operating System Interface, 2015) denominada *pthread.h*.

No presente trabalho, para fins de validação como mencionado anteriormente, na metodologia proposta utilizamos, além do verificador ESBMC++, os verificadores de modelo como LLBMC e DIVINE, os dois utilizam como base principal dos seus processos de verificações, a tradução do código-fonte a ser verificado em uma representação intermediária denominada *low level virtual machine* (LLVM). Deste modo, o modelo operacional simplificado multi-tarefas Qt proposto é utilizado no processo de tradução dos verificadores de modelos, para uma linguagem intermediária que os verificadores são capazes de reconhecer, possibilitando, assim, que os verificadores sejam capazes de verificar os programas concorrentes Qt, podendo validar se as propriedades da linguagem alvo da verificação estão sendo, de fato, atendidas.

Para realizar a validação, foi necessária uma comparação com relação a portabilidade e desempenho dos verificadores de modelos ESBMC++ e LLBMC, que se baseiam em técnicas de verificação SMT para a validação de corretude, já o verificador de modelo DIVINE realiza a verificação de modelos com o uso de espaços de estados explícitos.

Os verificadores de modelos utilizados nos processos de verificações foram utilizados da seguinte forma:

1. Primeiro: Iniciamos o processo de verificação utilizando o verificador de modelos ESBMC++.

2. Segundo: Adicionamos o modelo operacional simplificado proposto ao verificador de modelos LLBMC, que com o auxílio do clang, compila o código fonte que é passado ao verificador, criando, assim, um *bytecode* do código passado ao verificador. Em seguida, a partir do arquivo *bytecode* gerado, identifica seus os parâmetros iniciais e realiza a verificação.

3. Terceiro: Aplicamos o modelo operacional simplificado desenvolvido, ao verificador de modelos DIVINE, que realiza um processo de compilação do modelo que propomos, seme-

lhante aos outros verificadores utilizados. Ele converte os códigos passados no momento da verificação, em *bytecode*, para em seguida aplicar suas estratégias de verificação, sobre o código gerado.

As intercalações necessitam ser definidas para cada uma das ferramentas, ou seja, o valor de $\langle$ *bound* $\rangle$ que pode variar de acordo com caso de teste executado. No entanto como mencionado na Seção 4.1, no processo de verificação de programas multi-tarefas, não são necessárias muitas mudanças de contextos para expor as falhas, com isso, para os testes executados, definimos como padrão o $\langle$ *bound* $\rangle$ em 8 para todos os verificadores utilizados.

Comparamos os resultados experimentais, da adição do modelo operacional simplificado multi-tarefas, aos verificadores de modelos ESBMC++ versão 2.0 (utilizamos o Boolector como o principal solucionador SMT no processo de análise dos resultados por ter obtido os melhores resultados), LLBMC e o DIVINE. Dos três verificadores de modelos, aos quais o modelo operacional proposto foi adicionado, o ESBMC++ utilizando o Boolector como solucionador, foi o único a não apresentar estouro de memória ou falhas de segmentação, em todos os *benchmarks* utilizados.

Na Tabela 4, *TC* corresponde ao número de programas Qt/C++ multi-tarefas, *L* corresponde a quantidade de linhas de código, *Tempo* representa o tempo total de verificação para cada *benchmark*, *V* corresponde ao número de *benchmark* onde não foram encontrados defeitos, *F* corresponde ao número de *benchmark* em que foram detectadas falhas, *FP* corresponde ao número falsos positivos (ou seja, as ferramentas apontam como correto os casos de testes, que contêm erros e que deveriam falhar durante os processos de verificações). Durante a construção das suítes de teste, foram desenvolvido pelo menos um caso de teste positivo(como mencionado na Seção 5.1), no qual o programa não contém erros de implementação e um caso de teste negativo com alguma falha, para que o verificador de modelos possa identificar a falha no decorrer da verificação, em todas as suítes de testes desenvolvidas para testes. *FN* corresponde ao número de falsos negativos (ou seja, as ferramentas apontam falhas em casos de testes em que não têm a presença de falhas) e *Fail* representa o número de erros internos obtidos durante a verificação (e.g., erros de análise).

Na Tabela 4, é possível observar os resultados obtidos da comparação entre o ESBMC++ v1.25.4 (usando Boolector como solucionador SMT), LLBMC v2013.1 e DIVINE v3.3.2. Das verificações realizadas com o ESBMC++, apenas 2 dos 120 *benchmarks* resultaram em falsos

Tabela 4 – Tabela de resultados- Verificadores de modelos

Suíte de teste	TC	L	ESBMC++						LLBMC						DIVINE				
			Tempo	V	F	FP	FN	Fail	Tempo	V	F	FP	FN	Fail	Tempo	V	F	FP	FN
Qthread::Start()	30	960	110.76	15	14	1	0	0	80,15	13	14	2	1	0	1329.12	13	11	5	1
Qthread::run()	30	960	115,17	14	13	1	2	0	70,78	15	14	0	0	1	1382.04	15	10	4	1
Qthread::exec()	4	128	31.64	2	2	0	0	0	22,45	2	2	0	0	0	379.68	2	2	0	0
Qthread::quit()	4	130	40,45	2	2	0	0	0	33,14	2	2	0	0	0	485.4	2	2	0	0
Qthread::terminit()	4	110	25,45	2	2	0	0	0	12,12	2	2	0	0	0	305.4	2	2	0	0
Qthread::Priority()	16	512	67,569	8	8	0	0	0	75.569	7	0	8	0	1	810.828	8	4	2	2
LowestPriority()	4	128	31.64	2	2	0	0	0	221.48	2	2	0	0	0	411.32	2	2	0	0
LowPriority()	4	128	35,45	2	2	0	0	0	283.6	2	2	0	0	0	425.4	2	2	0	0
NormalPriority()	4	128	40,54	2	2	0	0	0	324.32	2	2	0	0	0	608.1	2	0	0	0
HighPriority()	4	128	56,44	2	2	0	0	0	395.08	2	2	0	0	0	677.28	2	0	0	0
HighestPriority()	4	128	31,55	2	2	0	0	0	220.85	2	2	0	0	0	410.15	2	2	0	0
TimeCriticalPriority()	4	128	32,48	2	2	0	0	0	259.84	2	2	0	0	0	389.76	2	2	0	0
InheritPriority()	4	128	33,48	2	2	0	0	0	234.36	2	2	0	0	0	401.76	2	2	0	0
Total	120	3696	652,61	49	47	2	15	0	2233.739	55	48	10	1	2	8016.238	56	49	11	4

positivos durante o processo de verificação das suítes de teste. Esses falsos positivos (FP) ocorrem quando o verificador não é capaz de verificar de forma correta os casos de teste negativos (suítes de testes que contém erros) *benchmarks*. O verificador DIVINE apresentou 11 *benchmarks* apontando falsos positivos, já o verificador LLBMC apresentou 10 *benchmarks* apontando falsos positivos, dos 120 *benchmarks* analisados. Isso ocorre, em grande parte, porque os verificadores não são capazes de verificar de forma correta o uso de objetos de sincronização condicional, que ocorrem por intermédio de objeto de sincronia (variáveis de condição), que estão implementadas no Modelo Operacional Simplificado Multi-Tarefas Qt/C++.

Na Figura 26, é possível observar os tempos de verificações de cada uma das suítes de testes, desenvolvidas para a validação dos métodos.

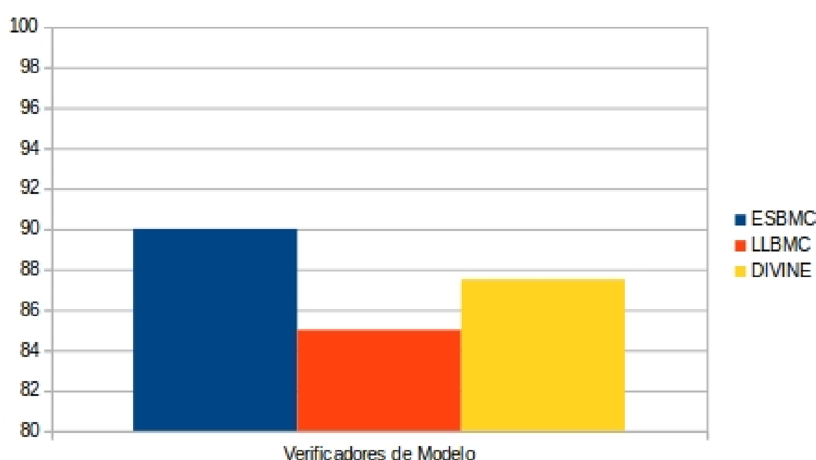


Figura 26 – Comparação entre os tempos de verificação das suítes de testes, resultantes da verificação do modelo operacional simplificado com os verificadores de modelos ESBMC++, LLBMC e DIVINE.

Entretanto, é importante ressaltar que o modelo operacional desenvolvido não cobre completamente todos os comportamentos que estão descritos na documentação do Qt. Ainda, assim, a combinação do modelo operacional proposto com os verificadores de modelos os tornaram

mais robustos, uma vez que por intermédio da adição do modelo proposto os verificadores BMC, mesmo apresentando lacunas no suporte à verificação da biblioteca *pthread*, mostraram-se capazes de realizar a verificação da corretude das principais estruturas concorrentes que foram modeladas do *framework* Qt.

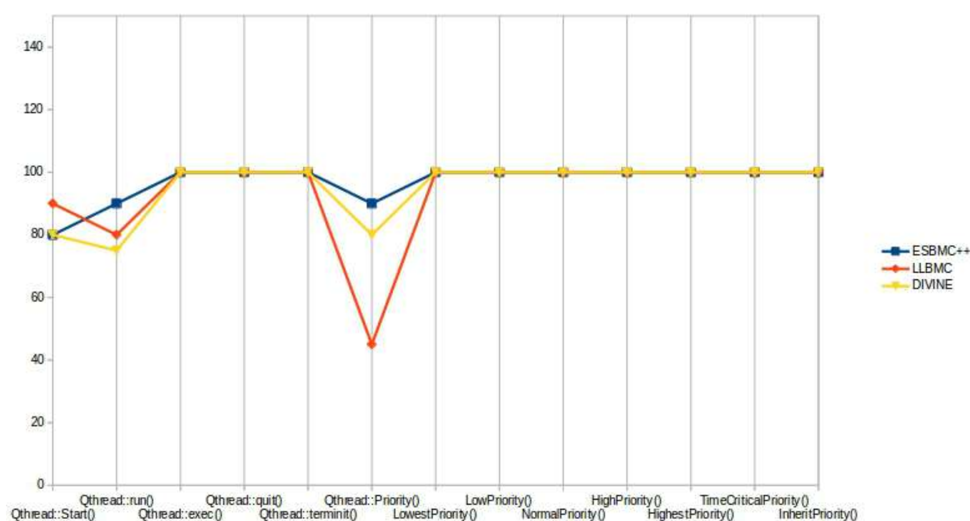


Figura 27 – Comparação entre as taxas de verificação das suítes de testes, resultantes da verificação do modelo operacional simplificado com os verificadores de modelos ESBMC++, LLBMC e DIVINE.

Na Figura 27, podemos observar que as taxas de verificações bem sucedidas, em todos os verificadores de modelos utilizados no processo de validação do modelo, ficaram acima de 95% para todas as suítes de testes analisadas, com relação aos casos de criação, execução e finalização das *threads*. Contudo, os verificadores LLBMC e o DIVINE, não mantiveram uma boa taxa de verificações bem sucedidas, ao analisar as suítes referentes aos casos de teste para analisar o sistema de prioridades, devido à inviabilidade dos verificadores de assegurar a confiabilidade da estrutura *pthread_cond_signal()*. É importante ressaltar que o conjunto de casos de testes presentes nas suítes *InheritPriority()*, *TimeCriticalPriority()*, *HighestPriority()*, *HighPriority()*, *NormalPriority()*, *NormalPriority()*, *NormalPriority()*, *Qthread::exec()*, *Qthread::quit()* e *Qthread::terminit()* tiveram uma taxa de verificações bem sucedidas de 100%, das suítes de testes analisadas.

Com isso, podemos constatar que as taxas de verificações bem sucedidas, dos verificadores de modelos utilizados para validar o modelo operacional, abrange em torno de 80% a 90% dos casos de teste analisados, o que demonstrou a eficiência do modelo operacional que propomos. Vale lembrar que cada caso de teste válida uma pré e pós-condição inerente à estrutura que está sendo validada. Como pode ser visto na Figura 27, o verificador ESBMC foi capaz de validar

cerca de 90% dos casos de teste avaliados, o verificador LLBMC obteve uma taxa de verificações bem sucedidas de 85% dos casos de teste avaliados e o verificador de modelos DIVINE obteve uma taxa de verificações bem sucedidas de 87.5% dos casos de teste avaliados, o que demonstra a eficácia do Modelo Operacional Simplificado Multi-Tarefas Qt/C++ proposto.

Podemos observar que o verificador de modelos ESBMC obteve os melhores resultados em relação aos verificadores de modelos LLBMC e DIVINE. Como conseguinte, o método que propomos, não é limitado a apenas uma ferramenta, podendo ser inserido em ferramentas de verificações de modelos, que mais se adaptam à abordagem de verificação do algoritmo que será verificado.

5.4 Avaliação Experimental - Análise do Modelo Operacional

5.4.1 Escolhas Determinísticas e Contra-Exemplos

O comportamento de um programa multi-tarefas que está em execução, depende muitas vezes de influências externas comumente chamadas de ambiente, que não podem ser descritas de forma determinística. O comportamento do ambiente, assim como seu efeito, imprimem um comportamento não determinístico no programa. Uma fonte importante desse não determinismo ocorre durante o processo de intercalação das *threads*, ou mesmo de forma equivalente, em que a *thread* entrará em execução após ocorrer uma interrupção (BARANOVÁ et al., 2017).

No presente trabalho, todo o não determinismo do programa e no sistema operacional é derivado do uso de *heap* de chamadas (áreas da memória disponibilizada para cada *thread*, que conta com um espaço reservado para o armazenamento das variáveis e dados que foram criados durante a execução do programa), que retorna entre 0 e um determinado número.

O que se mostrou uma abordagem muito interessante, como pode ser observado no verificador de modelos DIVINE, que também é utilizado no processo de rastreio do *heap* de chamadas para realizar rastreio de informações na pilha de execução das *threads*, com o intuito de anexar informações adicionais as transições do grafo gerado no processo de execução das verificações.

Em particular, as informações contidas no *heap* tornam-se parte do contra exemplo quando é apresentada ao usuário. Por exemplo, a *libc* (BARNAT et al., 2013) fornecida pelo DIVINE usa Hiper chamadas de Rastreio (servem para anexar informações adicionais às transições no

gráfico de execução) na implementação de funções de E/S padrão do verificador. Dessa forma, se um programa imprimir algo em sua saída padrão durante a execução da violação, essa saída se tornará visível no contra exemplo (ROČKAI et al., 2018).

5.4.2 Condições de Verificação

Condições de verificação são fórmulas lógicas, que são construídas partindo de um programa e de propriedades de corretude inerentes da linguagem alvo da verificação, onde sua validade pode ser comprovada pela verificação, se um determinado comportamento do programa está de acordo com a especificação da linguagem alvo da verificação. As propriedades de corretude, podem ser especificadas por intermédio de assertivas, que podem ser inseridas pelo usuário ou geradas automaticamente durante a verificação. De tal forma que, se todos as CVs de um programa foram validados pelas assertivas, o programa estará em conformidade com a sua especificação.

5.4.3 Pré-condições - Verificação dos Resultados Para os *benchmarks*

Como já mencionado na Seção 4, a utilização de modelos operacionais abstratos, contendo todas as propriedades que são necessárias para a garantia da corretude das estruturas relacionadas aos métodos da biblioteca *QThread*, alvo da verificação, foi possível diminuir a complexidade que é associada às árvores IRep (CORDEIRO, 2011). Com isso, o verificador ESBMC++ conseguiu construir uma árvore IRep de forma rápida e menos dispendiosa e de baixa complexidade. Para a validação de propriedades das estruturas do modelo, usamos assertivas que foram integradas às representações dos métodos no modelo operacional, com o objetivo de detectar violações em relação ao uso incorreto das estruturas representadas pelo modelo.

Com a adição das assertivas, os verificadores BMC que possuem suporte a estruturas C++ e a verificação de lógica booleana, tornam-se capazes de realizar a verificação de forma satisfatória.

Podemos observar essas assertivas no fragmento de código da Figura 31, que pertence ao conjunto de *benchmarks*, disponibilizados pelo *framework* Qt (The Qt Company Ltd., 2015). Neste *benchmark*, o produtor grava dados no *buffer* até atingir o final do *buffer*, ponto no qual ele reinicia, sobrescrevendo os dados existentes. A *thread* consumidora lê os dados à medida que são produzidos e os grava no *standard error*.

```
1 const int DataSize = 100000;
2
3 const int BufferSize = 8192;
4 char buffer[BufferSize];
5
6 QWaitCondition bufferNotEmpty;
7 QWaitCondition bufferNotFull;
8 QMutex mutex;
9 int numUsedBytes = 0;
```

Figura 28 – Escopo das variáveis globais

As condições de espera tornam possível ter um nível mais alto de simultaneidade do que o que é possível apenas com mutexes. Se os acessos ao *buffer* fossem protegidos por um *mutex*, a *thread* consumidora não poderia acessar o *buffer* simultaneamente com a *thread* produtor. No entanto, não existe impedimento em ter as duas *threads* trabalhando em diferentes partes do *buffer* em paralelo.

Os *benchmarks* mostrados na Figura 31, engloba duas classes: A classe *Poducer* e *Consumer*, as classes herdam da biblioteca *Qthread*. O *buffer* (região da memória, local onde é realizado o armazenamento temporário dos dados, enquanto eles estão sendo manipulados pelas classes) usado para a comunicação entre as duas classes e os métodos usados para a sincronização, são variáveis globais. Na Figura 28, podemos observar o escopo da declaração das variáveis globais, são elas: *DataSize* que corresponde a quantidade de dados que o produtor irá gerar (para manter a simplicidade do exemplo, declaramos a variável como uma constante) e *BufferSize* que corresponde ao tamanho do *buffer*. É importante observar que o valor atribuído ao tamanho do *BufferSize* é menor do que o atribuído ao *DataSize*, porque em algum momento o produtor irá alcançar o limite do *buffer*, quando isso ocorrer o *buffer* será reinicializado.

Para sincronizar o produtor e o consumidor, precisamos de duas condições de espera e um *mutex*. A condição *bufferNotEmpty* é sinalizada quando o produtor gerou alguns dados, informando ao consumidor que pode começar a lê-lo. A condição *bufferNotFull* é sinalizada quando o consumidor lê alguns dados, informando ao produtor que pode gerar mais. O *numUsedBytes* é o número de bytes no buffer que contém dados.

Juntas, as condições de espera, o *mutex* e o contador *numUsedBytes* garantem que o produtor nunca tenha mais do que os bytes *BufferSize* à frente do consumidor e que o consumidor nunca leia dados que o produtor ainda não gerou.

O produtor gera bytes de dados do *DataSize*. Antes de gravar um byte no buffer circular, ele deve primeiro verificar se o buffer está cheio (isto é, *numUsedBytes* é igual a *BufferSize*). Se o

```
1 class Producer : public QThread
2 {
3 public:
4     Producer(QObject *parent = NULL) : QThread(parent)
5     {
6     }
7
8     void run() override
9     {
10        for (int i = 0; i < DataSize; ++i) {
11            mutex.lock();
12            if (numUsedBytes == BufferSize)
13                bufferNotFull.wait(&mutex);
14            mutex.unlock();
15
16            buffer[i % BufferSize] = QRandomGenerator::global
                  ()->bounded(4);
17
18            mutex.lock();
19            ++numUsedBytes;
20            bufferNotEmpty.wakeAll();
21            mutex.unlock();
22        }
23    }
24 };
```

Figura 29 – Escopo do código para a classe *Producer*.

buffer estiver cheio, a *thread* aguarda na condição *bufferNotFull*.

No final, o produtor incrementa *numUsedBytes* e sinaliza que a condição *bufferNotEmpty* é verdadeira, já que *numUsedBytes* é necessariamente maior do que 0.

Nós guardamos todos os acessos à variável *numUsedBytes* com um mutex. Além disso, a função *QWaitCondition::wait()* aceita um *mutex* como seu argumento. Esse *mutex* é desbloqueado antes que a *thread* seja colocada em suspensão e bloqueada quando a *thread* é ativada. Além disso, a transição do estado bloqueado para o estado de espera é atômica, para evitar que as condições de corrida ocorram.

O código da Figura 29 proposto é muito semelhante ao exemplo do produtor. Antes de lermos o byte, verificamos se o buffer está vazio (*numUsedBytes* é 0) e aguardar a condição *bufferNotEmpty*, caso esteja vazio. Depois de lermos o byte, decrementamos *numUsedBytes* (em vez de incrementá-lo) e sinalizamos a condição *bufferNotFull* (em vez da condição *bufferNotEmpty*).

Em *main()*, criamos os dois encadeamentos e chamamos *QThread::wait()* para garantir que ambos os encadeamentos tenham tempo para serem concluídos antes de sairmos.

Entretanto, é importante notar que durante o processo de verificação, existem métodos e

```
1  class Consumer : public QThread
2  {
3    Q_OBJECT
4    public:
5      Consumer(QObject *parent = NULL) : QThread(parent)
6      {
7      }
8      void run() override
9      {
10       for (int i = 0; i < DataSize; ++i) {
11         mutex.lock();
12         if (numUsedBytes == 0)
13           bufferNotEmpty.wait(&mutex);
14         mutex.unlock();
15         cout >> buffer[i % BufferSize >> endl;
16         mutex.lock();
17         --numUsedBytes;
18         bufferNotFull.wakeAll();
19         mutex.unlock();
20       }
21     }
22 signals:
23   void stringConsumed(const QString &text);
24 };
```

Figura 30 – Escopo do código para a classe Consumer

funções, cuja finalidade é de imprimir os valores das variáveis. São interfaces, métodos re-implementáveis e funções sinalizadoras para objetos de sistema, as quais são implementadas no modelo apenas as assinaturas dos métodos, para que as estruturas possam ser reconhecidas pelo verificador de modelos BMC durante seus processos de verificação, possibilitando assim que seja construída a árvore IRep de forma confiável. Como podemos observar na Figura 32, alguns segmentos de código representam estruturas que não podem ser representados por proposições atômicas, o que impossibilita a extração das propriedades a serem validadas.

Inicialmente, a thread produtor é o único que pode fazer qualquer coisa; o consumidor está bloqueado esperando que a condição bufferNotEmpty seja sinalizada (numUsedBytes é 0). Uma vez que o produtor colocou um byte no buffer, numUsedBytes é BufferSize - 1 e a condição bufferNotEmpty é sinalizada. Nesse ponto, duas coisas podem acontecer: a *thread* consumidor assume e lê esse byte, ou o produtor consegue produzir um segundo byte.

O modelo produtor-consumidor apresentado neste exemplo possibilita a criação de aplicativos multi-tarefas altamente simultâneos. Em uma máquina multiprocessador, o programa é potencialmente até duas vezes mais rápido que o programa baseado em mutex equivalente, pois as duas *thread* podem estar ativas em paralelo em partes diferentes do buffer.

```
1
2 int main ( int argc , char * argv [ ]
    )
3 {
4     QCoreApplication app(argc, argv);
5     Producer producer;
6     Consumer consumer;
7     producer.start();
8     consumer.start();
9     producer.wait();
10    consumer.wait();
11    return 0;
12 }
```

Figura 31 – Representação do main()

Note que esses benefícios nem sempre são realizados. Bloquear e desbloquear um QMutex tem um custo. Na prática, provavelmente valeria a pena dividir o *buffer* em partes e operar em partes em vez de bytes individuais. O tamanho do *buffer* também é um parâmetro que deve ser selecionado com cuidado, com base na experimentação.

```
1 bool event(QEvent *event);
2 explicit QThread(QObject *parent = 0);
3 QThread(QThreadPrivate &dd, QObject *parent = 0);
4 bool wait(unsigned long time = ULONG_MAX);
```

Figura 32 – Representação de um método reimplementado de QObject

5.5 Avaliação de Conformidade do Operacional Simplificado Multi-Tarefas Qt/C++

Todos os aspectos de implementação, como processos, funções, APIs bem como as ferramentas e especificações necessárias para a implementação de aplicações que são desenvolvidas por intermédio do *framework* Qt, estão disponibilizadas por *The QT Developers* (COMPANY,).

No processo de desenvolvimento do modelo operacional simplificado, devemos considerar que é a partir do modelo operacional proposto que os verificadores BMC realizam a análise das estruturas a serem validadas, com isso é de extrema importância garantir a corretude das estruturas que foram desenvolvidas no modelo, objetivando assim a garantia do máximo de

exatidão no processo de verificação, com o intuito de obter a maior precisão no processo de validação.

Para que seja possível garantir a confiabilidade e o máximo de corretude dos testes realizados, o comportamento do modelo operacional simplificado desenvolvido deve se aproximar ao máximo do comportamento dos módulos analisados do *framework* Qt.

Visando a garantia da corretude e a confiabilidade do modelo proposto, durante o processo de desenvolvimento, realizamos uma análise de todas as suítes de teste implementadas no modelo, objetivando a garantia de corretude e confiabilidade dos *benchmarks* analisados. Todo o conjunto de suítes de teste foram desenvolvidos utilizando a linguagem C++, para reprodução e validação das estruturas analisadas do *framework* Qt. Buscamos analisar todas os possíveis comportamentos que tais estruturas podem assumir no decorrer de suas execuções, objetivando estar sempre em conformidade com a *The Qt Documentation* (COMPANY,).

Cada umas das funções foram analisadas com o intuito de validar as pré-condições e pós-condições que a estrutura de dados analisada deve assumir no decorrer de sua execução. Sendo assim, cada uma das funções analisadas tem pelo menos um caso de teste positivo (caso em que o resultado da verificação deve ser bem sucedida, pois não há erros no código a ser validado) e um caso de teste negativo (caso em que o resultado da verificação necessita falhar, por ter algum erro no código).

É importante ressaltar que todos os casos de teste presentes nas suítes de verificação foram desenvolvido e compilados na interface de desenvolvimento do *framework* Qt, chamada de *QtCreator*, objetivando que o modelo seja capaz de desenvolver um comportamento similar ao dos módulos analisados. O *QtCreator* é um ambiente de desenvolvimento integrado (IDE) multi-plataforma utilizado pelos desenvolvedores para criar diversas aplicações que utilizam o *framework* Qt para *desktop*, sistemas embarcados e plataformas de dispositivos móveis. É um ambiente de desenvolvimento disponível para Linux, OSX e Windows (The Qt Company Ltd., 2015).

Utilizamos também o compilador GCC (que é um compilador *front-end* para linguagens C e C++) (GNU Compiler Collection, 1987). Durante o processo de validação, compilamos o modelo operacional desenvolvido com o compilador GCC, objetivando analisar o comportamento do modelo, podendo assim verificar se as especificações do *framework* Qt estão sendo corretamente atendidas. Com isso, foi possível analisar o nível de conformidade do modelo desenvolvido com o *framework* Qt. Podemos observar essa comparação a partir do diagrama de atividades

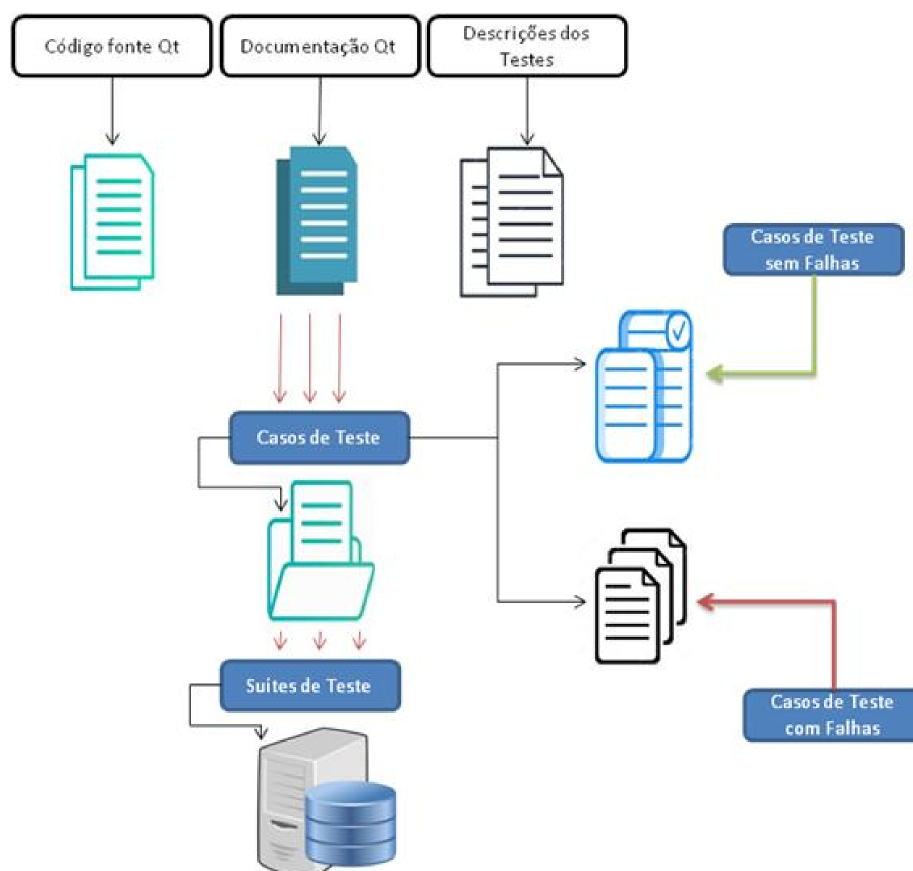


Figura 33 – Diagrama de atividades desenvolvidas no desenvolvimento do modelo operacional simplificado multitarefas QT.

mostrado na Figura 33.

É possível observar na Figura 34, o resultado da comparação, onde os casos de teste estão divididos em suítes de testes. É importante observar que a interface *QtCreator* é o ambiente de desenvolvimento padrão do *framework* Qt. Com isso, podemos alcançar um nível de conformidade de 100%, em todos os conjuntos de casos de teste que compõem o modelo operacional simplificado desenvolvido, buscando sempre a validação de todos os possíveis comportamentos sempre em conformidade com a *The Qt Documentation* (COMPANY,).

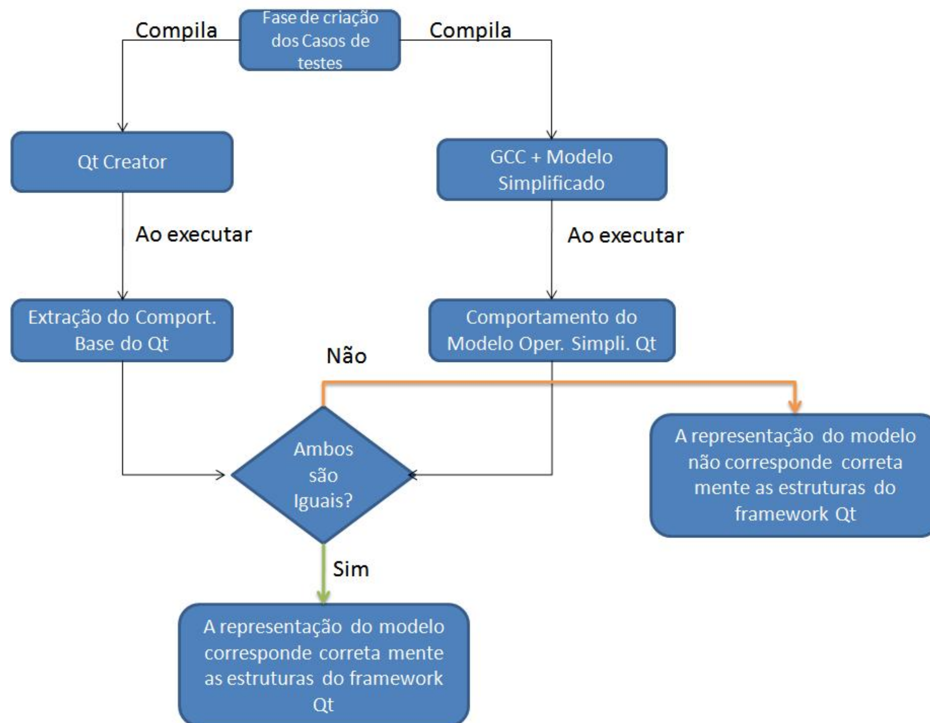


Figura 34 – Diagrama de atividades desenvolvidas para a validação da conformidade do Modelo Operacional Simplificado Multi-Tarefas QT.

6 Conclusões

A método proposta pela presente pesquisa, tem por objetivo a verificação de estruturas de dados multi-tarefas, desenvolvidas no *framework* Qt, sendo desenvolvido totalmente em Qt/C++, a qual é composta por pré-condições, pós-condições e verificações de propriedades presentes nos programas do *framework* Qt, que implementem sistemas multi-tarefas. Descrevemos ainda, a maneira com a qual desenvolvemos a implementação do Modelo Operacional Simplificado Multi-Tarefas Qt/C++, levando em consideração as técnicas necessárias para tornar possível a verificação de programas multi-tarefas, utilizando verificadores de modelos baseados em BMC.

Desenvolvemos suítes de testes para validar a metodologia abordada, por intermédio do verificador de modelos ESBMC++. Desta forma pudemos demonstrar a viabilidade da metodologia proposta para a verificação de aplicações multi-tarefas que utilizem o *framework* de desenvolvimento Qt.

O presente trabalho tem como principal contribuição o suporte a biblioteca *Qthreads*, que é a biblioteca principal de criação, manipulação e término das *threads*, presente no *framework* Qt, com a utilização do modelo operacional simplificado da biblioteca que foi desenvolvido, buscando validar todas as características e possíveis condições que o programa pode assumir no decorrer da verificação, tendo como base principal para a extração de propriedades da documentação do Qt (COMPANY,).

Validamos ainda o desempenho dos solucionadores Boolector, Yices 2 e Z3, uma vez que os mesmos foram utilizados no processo de verificação das suítes de teste. Os três solucionadores são nativos do ESBMC++ que foi integrado ao modelo operacional simplificado multi-tarefas desenvolvido. Considerando os resultados obtidos, o solucionador Boolector apresentou a maior taxa de cobertura dos programas analisados, tendo ainda o menor tempo de verificação, comparado aos outros solucionadores. Com a utilização do ESBMC++, obtivemos uma taxa de sucesso de aproximadamente 90%, para o conjunto de suítes de testes analisados. Pudemos analisar ainda, o nível de conformidade no modelo desenvolvido comparando o seu comportamento com o comportamento do *framework* Qt. Tal comparação, resultou em um nível de conformidade entre todas as suítes de teste analisadas de aproximadamente 90% .

Por fim, mais não menos importante, outra importante contribuição do modelo de operacional proposto, é a possibilidade de integrá-lo de forma bem sucedida aos processos de verificações

de verificadores de modelos que possuem suporte à verificação de programas C++. Como mencionamos durante o desenvolvimento desta pesquisa, utilizamos os verificadores LLBMC e o verificador DIVINE, o que pode comprovar a sua versatilidade para a validação da metodologia que abordamos. A utilização do modelo operacional simplificado multi-tarefas ao ser integrado ao processo de verificação do verificador LLBMC foi capaz de detectar 85% dos erros existentes. No entanto, foi o verificador que obteve o maior tempo de verificação das suítes de testes. Já o verificador de modelos DEVINE foi capaz de detectar em média 87.5% dos erros existentes. O DIVINE foi o verificador com o menor tempo no processo de verificação das suítes de teste.

É importante, ressaltar que, o verificador de modelos DIVINE obteve a maior taxa de resultados incorretos, cerca de 15%, seguido pelo verificador de modelos LLBMC com 13%. Já o verificador de modelos ESBMC++, obteve o menor número de resultados incorretos durante o processo de verificação, com uma taxa de erro de 5%. Com isso, o modelo operacional simplificado multi-tarefas pode ser integrado a ferramentas de verificação que possuem suporte a verificação de sistemas C++ e a biblioteca *pthreads*, usada para a modelagem e a manipulação das *threads* dentro do modelo operacional, como podemos comprovar, pelo uso dos verificados ESBMC++, DEVINE e LLBMC.

6.1 Trabalhos Futuros

Como trabalhos futuros, propomos estender o modelo operacional simplificado multi-tarefas desenvolvido, incluído mais bibliotecas do *framework* Qt (isto é, incluindo mais classes e bibliotecas), com o objetivo de aumentar a cobertura do modelo, aumentando assim a capacidade de verificação das estruturas analisadas e dessa maneira tornar possível a validação de um conjunto maior de propriedades pertencentes ao *framework* Qt, bem como aumentar a quantidade de *benchmarks* utilizados para validação do modelo, além de aplicar o Modelo Operacional Simplificado Multi-Tarefas Qt/C++ em uma ferramenta de grande porte de utilização do mundo real. Pretendemos também buscar novas ferramentas de verificação de modelos, para programas C++ e multi-tarefas, com o intuito de validar a metodologia abordada.

6.2 Agradecimentos

Esta pesquisa, conforme previsto no Art. 48 do decreto nº 6.008/2006, foi financiada pela Samsung Electrônica da Amazônia Ltda, nos termos da Lei Federal nº 8.387/1991, através de convênio nº 004, firmado com o CETELI/ UFAM.

Referências

PIPATSRISAWAT, K.; DARWICHE, A. On the power of clause-learning sat solvers with restarts. In: SPRINGER. *International Conference on Principles and Practice of Constraint Programming*. 2009. p. 654–668.

GADELHA, M. Y. R. et al. Verificação baseada em indução matemática para programas c++. Universidade Federal do Amazonas, 2013.

COMPANY, T. Q.

CORDEIRO, L. C. Smt-based bounded model checking for multi-threaded software in embedded systems. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 2010. p. 373–376.

CORDEIRO, L. C.; FISCHER, B. Verifying multi-threaded software using smt-based context-bounded model checking. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 2011. p. 331–340.

CORDEIRO, L. *SMT-based bounded model checking of multi-threaded software in embedded systems*. Tese (Doutorado) — University of Southampton, 2011.

The Qt Company Ltd. *The Qt Framework*. 2015. <http://www.qt.io/qt-framework/>. [Online; accessed 2-April-2015].

SOUSA, F. R. M.; CORDEIRO, L. C.; FILHO, E. B. de L. Bounded model checking of C++ programs based on the qt framework. In: *IEEE 4th Global Conference on Consumer Electronics, GCCE 2015, Osaka, Japan, 27-30 October 2015*. 2015. p. 179–180.

GARCIA, M.; MONTEIRO, F. R.; CORDEIRO, L. C.; FILHO, E. B. de L. Esbmc^{qtom}: A bounded model checking tool to verify qt applications. In: *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*. : Springer, 2016. (Lecture Notes in Computer Science, v. 9641), p. 97–103.

MONTEIRO, F. R.; GARCIA, M.; CORDEIRO, L. C.; FILHO, E. B. de L. Bounded model checking of C++ programs based on the qt cross-platform framework. *Softw. Test., Verif. Reliab.*, v. 27, n. 3, 2017.

MONTEIRO, F. R.; GARCIA, M. A. P.; CORDEIRO, L. C.; FILHO, E. B. de L. Bounded model checking of C++ programs based on the qt cross-platform framework (journal-first abstract). In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 2018. p. 954.

BEYER, D. Software verification and verifiable witnesses. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2015. p. 401–416.

PEREIRA, P. A.; ALBUQUERQUE, H. F.; MARQUES, H.; SILVA, I. da; CARVALHO, C.; CORDEIRO, L. C.; SANTOS, V.; FERREIRA, R. Verifying CUDA programs using smt-based context-bounded model checking. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. 2016. p. 1648–1653.

PEREIRA, P. A.; ALBUQUERQUE, H. F.; SILVA, I. da; MARQUES, H.; MONTEIRO, F. R.; FERREIRA, R.; CORDEIRO, L. C. Smt-based context-bounded model checking for CUDA programs. *Concurrency and Computation: Practice and Experience*, v. 29, n. 22, 2017.

MONTEIRO, F. R.; ALVES, E. H. da S.; SILVA, I. da; ISMAIL, H.; CORDEIRO, L. C.; FILHO, E. B. de L. ESBMC-GPU A context-bounded model checking tool to verify CUDA programs. *Sci. Comput. Program.*, v. 152, p. 63–69, 2018.

ALVES, E. H. da S. Localização de falhas em programas concorrentes em c. 2018.

ROCHA, H.; BARRETO, R. S.; CORDEIRO, L. C.; NETO, A. D. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*. : Springer, 2012. (Lecture Notes in Computer Science, v. 7321), p. 128–142.

ALVES, E. H. da S.; CORDEIRO, L. C.; FILHO, E. B. de L. Fault localization in multi-threaded C programs using bounded model checking. In: *2015 Brazilian Symposium on Computing Systems Engineering, SBESC 2015, Foz do Iguacu, Brazil, November 3-6, 2015*. 2015. p. 96–101.

ALVES, E. H. da S.; CORDEIRO, L. C.; FILHO, E. B. de L. A method to localize faults in concurrent C programs. *Journal of Systems and Software*, v. 132, p. 336–352, 2017.

GADELHA, M. Y. R.; MONTEIRO, F. R.; MORSE, J.; CORDEIRO, L. C.; FISCHER, B.; NICOLE, D. A. ESBMC 5.0: an industrial-strength C model checker. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 2018. p. 888–891.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. : John Wiley & Sons, 2011.

JANUÁRIO, F. A. P.; CORDEIRO, L. C.; JR., V. F. de L.; FILHO, E. B. de L. Bmclua: Verification of lua programs in digital TV interactive applications. In: *IEEE 3rd Global Conference on Consumer Electronics, GCCE 2014, Tokyo, Japan, 7-10 October 2014*. 2014. p. 707–708.

MONTEIRO, F. R.; JANUÁRIO, F. A. P.; CORDEIRO, L. C.; FILHO, E. B. de L. Bmclua: A translator for model checking lua programs. *ACM SIGSOFT Software Engineering Notes*, v. 42, n. 3, p. 1–10, 2017.

SOUZA, A. S. de; CORDEIRO, L. C.; JANUARIO, F. A. Verificação de programas multi-thread baseados no framework multiplataformas qt. 2018.

GANAI, M. K.; GUPTA, A. Completeness in smt-based bmc for software programs. In: ACM. *Proceedings of the conference on Design, automation and test in Europe*. 2008. p. 831–836.

The Qt Company Ltd. Documentation. *Qthread.h*. 2015. <https://doc.qt.io/qt-5/qthread.html>. [Online; accessed 2-Janeiro-2016].

CORDEIRO, L. C.; MORSE, J.; NICOLE, D. A.; FISCHER, B. Context-bounded model checking with ESBMC 1.17 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012. p. 534–537.

MORSE, J.; CORDEIRO, L. C.; NICOLE, D. A.; FISCHER, B. Handling unbounded loops with ESBMC 1.20 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. : Springer, 2013. (Lecture Notes in Computer Science, v. 7795), p. 619–622.

MORSE, J.; RAMALHO, M.; CORDEIRO, L. C.; NICOLE, D. A.; FISCHER, B. ESBMC 1.22 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. : Springer, 2014. (Lecture Notes in Computer Science, v. 8413), p. 405–407.

BARNAT, J.; BRIM, L.; HAVEL, V.; HAVLÍČEK, J.; KRIHO, J.; LENČO, M.; ROČKAI, P.; ŠTILL, V.; WEISER, J. Divine 3.0–an explicit-state model checker for multithreaded c & c++ programs. In: SPRINGER. *International Conference on Computer Aided Verification*. 2013. p. 863–868.

SOUSA, F. R. M. Verificação formal de programas c++ que usam o framework multi-plataforma qt. Universidade Federal do Amazonas, 2013.

RAMALHO, M.; FREITAS, M.; SOUSA, F. R. M.; MARQUES, H.; CORDEIRO, L. C.; FISCHER, B. Smt-based bounded model checking of C++ programs. In: *20th IEEE International Conference and Workshops on Engineering of Computer Based Systems, ECBS 2013, Scottsdale, AZ, USA, April 22-24, 2013*. 2013. p. 147–156.

MAGERMAN, D. M. Statistical decision-tree models for parsing. In: ASSOCIATION FOR COMPUTATIONAL LINGUISTICS. *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*. 1995. p. 276–283.

MERZ, F.; FALKE, S.; SINZ, C. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In: SPRINGER. *International Conference on Verified Software: Tools, Theories, Experiments*. 2012. p. 146–161.

BARANOVÁ, Z.; BARNAT, J.; KEJSTOVÁ, K.; KUCERA, T.; LAUKO, H.; MRÁZEK, J.; ROCKAI, P.; STILL, V. Model checking of C and C++ with DIVINE 4. In: *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*. : Springer, 2017. (Lecture Notes in Computer Science, v. 10482), p. 201–207.

FALKE, S.; MERZ, F.; SINZ, C. The bounded model checker llbmc. In: IEEE. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013. p. 706–709.

LATTNER, C.; ADVE, V. The llvm compiler framework and infrastructure tutorial. In: SPRINGER. *International Workshop on Languages and Compilers for Parallel Computing*. 2004. p. 15–16.

Portable Operating System Interface. *Interface Portável entre Sistemas Operativos*. 2015. <https://standards.ieee.org/regauth/posix/>. [Online; accessed 2-Abril-2015].

BRADLEY, A. R.; MANNA, Z. *The calculus of computation: decision procedures with applications to verification*. : Springer Science & Business Media, 2007.

MOURA, L. de; BJØRNER, N. *Z3-a Tutorial*. : Citeseer, 2011.

BJØRNER, N.; PHAN, A.-D. νz-maximal satisfaction with z3. *Scss*, v. 30, p. 1–9, 2014.

DUTERTRE, B. Yices 2.2. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. : Springer, 2014. (Lecture Notes in Computer Science, v. 8559), p. 737–744.

DENKER, G.; KAGAL, L. Sri international. Citeseer, 1978.

BARRETT, C. "decision procedures: An algorithmic point of view,"by daniel kroening and ofer strichman, springer-verlag, 2008. *J. Autom. Reasoning*, v. 51, n. 4, p. 453–456, 2013.

TRINDADE, A. B. et al. Aplicando verificação de modelos baseada nas teorias do módulo da satisfabilidade para o particionamento de hardware/software em sistemas embarcados. Universidade Federal do Amazonas, 2015.

ALVES, E. H. d. S. et al. Localização de falhas em programas concorrentes em c. Universidade Federal do Amazonas, 2018.

BARRETT, C.; MOURA, L. M. de; RANISE, S.; STUMP, A.; TINELLI, C. The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In: *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*. : Springer, 2010. (Lecture Notes in Computer Science, v. 6504), p. 3.

GANESH, V.; DILL, D. L. A decision procedure for bit-vectors and arrays. In: SPRINGER. *International Conference on Computer Aided Verification*. 2007. p. 519–531.

CORDEIRO, L. C. Automated verification and synthesis of embedded systems using machine learning. *CoRR*, abs/1702.07847, 2017. Disponível em: <http://arxiv.org/abs/1702.07847>.

CORDEIRO, L. C.; FILHO, E. B. de L. Smt-based context-bounded model checking for embedded systems: Challenges and future trends. *ACM SIGSOFT Software Engineering Notes*, v. 41, n. 3, p. 1–6, 2016.

GADELHA, M. Y. R.; MONTEIRO, F. R.; CORDEIRO, L. C.; NICOLE, D. A. Towards counterexample-guided k-induction for fast bug detection. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. : ACM, 2018. p. 765–769.

BAIER, C.; KATOEN, J. *Principles of model checking*. : MIT Press, 2008. ISBN 978-0-262-02649-9.

CLARKE, E. M.; HENZINGER, T. A.; VEITH, H. Introduction to model checking. In: *Handbook of Model Checking*. 2018. p. 1–26.

HANSEN, P. B. Distributed processes: A concurrent programming concept. In: *The origin of concurrent programming*. : Springer, 1978. p. 444–463.

SANTOS, B. R. et al. Um método para verificação formal e dinâmica de sistemas de software concorrentes. Universidade Federal de Alagoas, 2016.

MCMILLAN, J. B. E. C. K.; HWANG, D. D. L. Symbolic model checking: 10 states and beyond. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science: June 4-7, 1990, Philadelphia, PA*. 1990. p. 428.

PRABHU, P.; MAEDA, N.; BALAKRISHNAN, G.; IVANČIĆ, F.; GUPTA, A. Interprocedural exception analysis for c++. In: SPRINGER. *European Conference on Object-Oriented Programming*. 2011. p. 583–608.

QADEER, S. Taming concurrency: A program verification perspective. In: *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*. : Springer, 2008. (Lecture Notes in Computer Science, v. 5201), p. 5.

DEJNOZKOVÁ, E.; DOKLÁDAL, P. Asynchronous multi-core architecture for level set methods. In: IEEE. *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*. 2004. v. 5, p. V–1.

RU, J.; KEUNG, J. An empirical investigation on the simulation of priority and shortest-job-first scheduling for cloud-based software systems. In: IEEE. *2013 22nd Australian Software Engineering Conference*. 2013. p. 78–87.

SCHRAGE, L. Letter to the editor—a proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, INFORMS, v. 16, n. 3, p. 687–690, 1968.

SHREEDHAR, M.; VARGHESE, G. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, IEEE, n. 3, p. 375–385, 1996.

ZAHARIA, M.; BORTHAKUR, D.; SARMA, J. S.; ELMELEEGY, K.; SHENKER, S.; STOICA, I. *Job scheduling for multi-user mapreduce clusters*. 2009.

KAY, J.; LAUDER, P. A fair share scheduler. *Communications of the ACM*, ACM, v. 31, n. 1, p. 44–55, 1988.

SANGIOVANNI-VINCENTELLI, A.; CARLONI, L.; BERNARDINIS, F. D.; SGROI, M. Benefits and challenges for platform-based design. In: ACM. *Proceedings of the 41st annual Design Automation Conference*. 2004. p. 409–414.

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: SPRINGER. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2008. p. 337–340.

BIERE, A.; CIMATTI, A.; CLARKE, E. M.; STRICHMAN, O.; ZHU, Y. et al. Bounded model checking. *Advances in computers*, v. 58, n. 11, p. 117–148, 2003.

ZAKS, A.; JOSHI, R. Verifying multi-threaded c programs with spin. In: SPRINGER. *International SPIN Workshop on Model Checking of Software*. 2008. p. 325–342.

HOLZMANN, G. J. The model checker spin. *IEEE Transactions on software engineering*, IEEE, v. 23, n. 5, p. 279–295, 1997.

HOLZMANN, G. J.; BOSNACKI, D. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, IEEE, v. 33, n. 10, p. 659–674, 2007.

INVERSO, O.; TOMASCO, E.; FISCHER, B.; TORRE, S. L.; PARLATO, G. Bounded model checking of multi-threaded c programs via lazy sequentialization. In: SPRINGER. *International Conference on Computer Aided Verification*. 2014. p. 585–602.

QADEER, S.; WU, D. Kiss: keep it simple and sequential. In: ACM. *Acm sigplan notices*. 2004. v. 39, n. 6, p. 14–24.

INVERSO, O.; TOMASCO, E.; FISCHER, B.; TORRE, S. L.; PARLATO, G. Lazy-cseq: a lazy sequentialization tool for c. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2014. p. 398–401.

CHO, C. Y.; D'SILVA, V.; SONG, D. Blitz: Compositional bounded model checking for real-world programs. In: IEEE PRESS. *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. 2013. p. 136–146.

ALGLAVE, J.; KROENING, D.; TAUTSCHNIG, M. Partial orders for efficient bounded model checking of concurrent software. In: SPRINGER. *International Conference on Computer Aided Verification*. 2013. p. 141–157.

GANAI, M. K.; GUPTA, A. Efficient modeling of concurrent systems in bmc. In: SPRINGER. *International SPIN Workshop on Model Checking of Software*. 2008. p. 114–133.

HATCLIFF, J.; DWYER, M. B. et al. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In: SPRINGER. *International Workshop on Verification, Model Checking, and Abstract Interpretation*. 2004. p. 175–190.

FLANAGAN, C.; GODEFROID, P. Dynamic partial-order reduction for model checking software. In: ACM. *ACM Sigplan Notices*. 2005. v. 40, n. 1, p. 110–121.

JR, T. W. D. Parallel program correctness through refinement. In: ACM. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977. p. 155–169.

FREUND, S. N.; QADEER, S. Checking concise specifications for multithreaded software. *Journal of Object Technology*, v. 3, n. 6, p. 81–101, 2004.

FLANAGAN, C.; SAXE, J. B. Avoiding exponential explosion: Generating compact verification conditions. In: ACM. *ACM SIGPLAN Notices*. 2001. v. 36, n. 3, p. 193–205.

DIJKSTRA, E. W.; DIJKSTRA, E. W.; DIJKSTRA, E. W.; INFORMATICIEN, E.-U.; DIJKSTRA, E. W. *A discipline of programming*. : prentice-hall Englewood Cliffs, 1976.

NELSON, C. G. *Techniques for program verification*. : Xerox. Palo Alto Research Center, 1981.

International Organization for Standardization. *A ISO é uma entidade de padronização e normatização, e foi criada em Genebra, na Suíça, em 1947*. 2015. <https://www.iso.org/home.html>. [Online; accessed 2-Maio-2015].

CORRÊA, E. d. F.; FRIEDRICH, L. F. Padronização posix x sistemas operacionais de tempo-real: uma análise comparativa. In: *III Congreso Argentino de Ciencias de la Computación*. 1997.

BAO, T.; JONES, M. Model checking abstract components within concrete software environments. In: SPRINGER. *International SPIN Workshop on Model Checking of Software*. 2008. p. 42–59.

GODEFROID, P. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In: *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996*. 1996. p. 289–304.

MCMILLAN, K. L. Symbolic model checking. In: *Symbolic Model Checking*. : Springer, 1993. p. 25–60.

GANAI, M. K.; GUPTA, A. Accelerating high-level bounded model checking. In: ACM. *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. 2006. p. 794–801.

QADEER, S.; REHOF, J. Context-bounded model checking of concurrent software. In: SPRINGER. *International conference on tools and algorithms for the construction and analysis of systems*. 2005. p. 93–107.

CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 4, p. 957–974, 2012.

MCMILLAN, K. L. Interpolants and symbolic model checking. In: SPRINGER. *International Workshop on Verification, Model Checking, and Abstract Interpretation*. 2007. p. 89–90.

LAL, A.; REPS, T. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, Springer, v. 35, n. 1, p. 73–97, 2009.

MERZ, S. Model checking: A tutorial overview. In: SPRINGER. *Summer School on Modeling and Verification of Parallel Processes*. 2000. p. 3–38.

CLARKE, E.; KROENING, D.; LERDA, F. A tool for checking ansi-c programs. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2004. p. 168–176.

CLARKE, E. M. Sat-based counterexample guided abstraction refinement in model checking. In: SPRINGER. *International Conference on Automated Deduction*. 2003. p. 1–1.

REHOF, J.; QADEER, S. Context-bounded model checking of concurrent software. In: CITESEER. *TACAS'05*. 2005. p. 93–107.

BIERE, A.; HELJANKO, K.; JUNTTILA, T.; LATVALA, T.; SCHUPPAN, V. Linear encodings of bounded ltl model checking. *arXiv preprint cs/0611029*, 2006.

TRETMANS, J. Model based testing with labelled transition systems. In: *Formal methods and testing*. : Springer, 2008. p. 1–38.

ROČKAI, P.; ŠTILL, V.; ČERNÁ, I.; BARNAT, J. Divm: Model checking with llvm and graph memory. *Journal of Systems and Software*, Elsevier, v. 143, p. 1–13, 2018.

MAZIERO, C. A. Sistemas operacionais: Conceitos e mecanismos. *Livro aberto. Acessível em: http://wiki. inf. ufpr. br/maziero/lib/exe/fetch. php*, p. 27, 2014.

BRUMMAYER, R.; BIERE, A. Boolector: An efficient smt solver for bit-vectors and arrays. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2009. p. 174–177.

JONES, C. B. Specification and design of (parallel) programs. In: *IFIP congress*. 1983. v. 83, p. 321–332.

GNU Compiler Collection. *GNU C Compiler (compilador C GNU)*. 1987. <https://gcc.gnu.org/>. [Online; accessed 2-April-2014].