Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Verificação de programas C++ baseados no *framework* cross-plataforma Qt

Mário Angel Praia Garcia

Manaus – Amazonas

Agosto de 2016

Mário Angel Praia Garcia

Verificação de programas C++ baseados no *framework* cross-plataforma Qt

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Orientador: Lucas Carvalho Cordeiro

Coorientador: Waldir Sabino da Silva Júnior

Mário Angel Praia Garcia

Verificação de programas C++ baseados no *framework* cross-plataforma Qt

Banca Examinadora

Prof. Ph.D. Lucas Carvalho Cordeiro – Orientador
Departamento de Eletrônica e Computação – UFAM

Prof. D.Sc. Waldir Sabino da Silva Júnior – Presidente e Coorientador
Departamento de Eletrônica e Computação – UFAM

Prof. D.Sc. Raimundo da Silva Barreto
Instituto de Computação – UFAM

Prof. D.Sc. Tayana Uchôa Conte
Instituto de Computação – UFAM

Manaus – Amazonas
Agosto de 2016

À minha mãe, avó e tias.

Agradecimentos

Agradeço primeiramente a Deus por ter iluminado meu caminho, minhas decisões e me proporcionado está oportunidade.

Agradeço à minha mãe, Fátima Tereza Praia Garcia, por estar sempre do meu lado e cujo amor é incondicional. As minhas tias, Letice e Lenice Praia, por me ajudarem em meu crescimento pessoal e incentivo a não desistir dessa jornada e à minha avó, Maria Tereza Praia Soares Lima, por sua paciência, compreensão, ajuda e pelo seu amor.

Agradeço aos meus amigos e colegas de mestrado da UFAM, que me ajudaram diretamente e inderetamente nessa jornada, pelo incentivo e apoio dado.

Agradeço aos amigos e ex-colegas de graduação da UEA, pela amizade e companheirismo.

Agradeço ao Prof. Lucas Cordeiro, pela dedicação, paciência, encorajamento e apoio oferecido para a realização desse trabalho.

Agradeço à equipe do projeto de verificação formal na UFAM, que contribuíram direta e indiretamente para a realização deste trabalho.

“Viver não é necessário. Necessário é criar.”.

Fernando Pessoa (1888-1935)

Resumo

O desenvolvimento de software para sistemas embarcados tem crescido rapidamente, o que na maioria das vezes acarreta em um aumento da complexidade associada a esse tipo de projeto. Como consequência, as empresas de eletrônica de consumo costumam investir recursos em mecanismos de verificação rápida e automática, com o intuito de desenvolver sistemas robustos e assim reduzir as taxas de *recall* de produtos. Além disso, a redução no tempo de desenvolvimento e na robustez dos sistemas desenvolvidos podem ser alcançados através de *frameworks* multi-plataformas, tais como Qt, que oferece um conjunto de bibliotecas (gráficas) confiáveis para vários dispositivos embarcados. Desta forma, este trabalho propõe uma versão simplificada do *framework* Qt que integrado a um verificador baseado nas teorias do módulo da satisfatibilidade, denominado *Efficient SMT-Based Bounded Model Checker* (ESBMC++), verifica aplicações reais que ultilizam o Qt, apresentando uma taxa de sucesso de 89%, para os *benchmarks* desenvolvidos. Com a versão simplificada do *framework* Qt proposto, também foi feita uma avaliação ultilizando outros verificadores que se encontram no estado da arte para verificação de programas em C++ e uma avalição a cerca de seu nível de conformidade. Dessa maneira, a metodologia proposta se afirma como a primeira a verificar formalmente aplicações baseadas no *framework* Qt, além de possuir um potencial para desenvolver novas frentes para a verificação de código portátil.

Palavras-chave: Modelo Operacional, *Framework* Qt, Verificação Formal, *Bounded Model Checking*.

Abstract

The software development for embedded systems is getting faster and faster, which generally incurs an increase in the associated complexity. As a consequence, consumer electronics companies usually invest a lot of resources in fast and automatic verification mechanisms, in order to create robust systems and reduce product recall rates. In addition, further development-time reduction and system robustness can be achieved through cross-platform frameworks, such as Qt, which favor the reliable port of software stacks to different devices. Based on that, the present work proposes a simplified version of the Qt framework, which is integrated into a checker based on satisfiability modulo theories (SMT), name as the efficient SMT-based bounded model checker (ESBMC++), for verifying actual Qt-based applications, and presents a success rate of 89%, for the developed benchmark suite. We also evaluate our simplified version of the Qt framework using other state-of-the-art verifiers for C++ programs and an evaluation about their level of compliance. It is worth mentioning that the proposed methodology is the first one to formally verify Qt-based applications, which has the potential to devise new directions for software verification of portable code.

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

Índice

Índice de Figuras

Índice de Tabelas

Abreviações

QtOM - ***O**perational **M**odel **Q**t*

IU - **I**nterface para **U**suário

MO - **M**odelo **O**peracional

BMC - ***B**ounded **M**odel **C**hecking*

SMT - ***S**atisfiability **M**odulo **T**heories*

ESBMC - ***E**fficient **S**MT-Based Context-**B**ounded **M**odel **C**hecker*

VC - ***V**erification **C**condition*

GPU - ***G**raphics **P**rocessing **U**nit*

SAT - ***B**oolean **S**atisfiability*

LLBMC - ***L**ow **L**evel **B**ounded **M**odel **C**hecker*

STL - ***S**tandard **T**emplate **L**ibrary*

CBMC - ***C B**ounded **M**odel **C**hecker*

GCC - ***GNU C**ompiler **C**ollection*

AST - ***A**bstract **S**yntax **T**ree*

VCG - ***V**erification **C**condition **G**enerator*

TMS - ***T**iled **M**ap **S**ervice*

IREP - ***I**ntermediate **REP**resentation*

SSA - ***S**ingle **S**tatic **A**ssignment*

GUI - ***G**raphical **U**ser **I**nterface*

LIFO - ***L**ast-in **F**irst-out*

FIFO - ***F**irst-in **F**irst-out*

IDE - ***I**ntegrated **D**evelopment **E**nvironment*

Capítulo 1

Introdução

A atual disseminação dos sistemas embarcados e sua devida importância está de acordo com a evolução dos componentes de hardware e software associados a eles [1]. Na realidade, esses sistemas têm crescido em relação a sua robustez e complexidade, onde se torna visível o uso de processadores com vários núcleos e memórias compartilhadas escaláveis, de maneira a suprir o crescimento do poder computacional exigido. Neste contexto, Qt apresenta-se como um poderoso *framework* multiplataforma para sistemas embarcados que enfatiza a criação de interface para usuário (IU) e o desenvolvimento de aplicações gráficas [2]. No entanto, a dependência do usuário em relação ao bom funcionamento de tais sistemas tem crescido de maneira rápida, assim como as suas complexidades. Em consequência, a confiabiliadade e a segurança de tais sistesmas torna-se algo de grande importância no processo de desenvolvimento de dispositivos comerciais e suas aplicações específicas.

Empresas de eletrônica de consumo cada vez investem mais em tempo e esforço para desenvolver alternativas rápidas e baratas referentes à verificação, com o objetivo de verificar a corretude de seus sistemas com o intuito de evitar perdas fincanceiras [3]. Entre as alternativas já existentes, uma das mais eficiente e menos custosa é a abordagem da verificação de modelos [4] [5]. Porém, existem muitos sistemas que não são possíveis de verificá-los de uma maneira automática devido à falta de suporte de certos tipos de linguagens e *frameworks* por parte dos verificadores de código. Por exemplo, o verificador *Java PathFinder* é capaz de verificar códigos em Java baseado em *bytecode* [6], mas não há suporte a verificação (completa) de aplicações Java que utilizam o sistema operacional Android [7]. Na realidade, esta verificação somente se torna possível se existir uma representação abstrata das bibliotecas associadas, de-

nominada de modelo operacional (MO) que, de forma conservadora, aproxima-se a semântica usada pelo sistema a ser verificado [8].

Essa dissertação identifica as principais características do *framework* Qt e propõe um modelo operacional (MO) que tem como propósito analisar e verificar as propriedades relacionadas de acordo com as suas funcionalidades. Vale ressaltar, que este trabalho é uma extensão de artigos já publicados anteriormente [9, 10] e os algoritmos desenvolvidos neste trabalho foram intregrados em uma ferramenta de verificação de modelos limitada (do inglês, *Bounded Model Checking* - BMC) baseada nas teorias do módulo da satisfatibilidade (do inglês, *satisfiability modulo theories* - SMT), denominada *Efficient SMT-based Context-Bounded Model Checker* (ESBMC++) [9] [11] [12], a fim de verificar propriedades específicas de programas escritos em Qt/C++.

A combinação entre ESBMC e MOs foi aplicada anteriormente para fornecer suporte à verificação de programas escritos em C++, conforme descrito por Ramalho *et al.* [9]. No entanto, no método proposta, um MO é utilizado para identificar elementos do *framework* Qt e verificar propriedades específicas relacionadas a estas estruturas por meio de pré e pós-condições [10].

1.1 Descrição do Problema

A dependência do usuário com relação ao funcionamento correto de sistemas embarcados tem aumentando rapidamente. Isto ocorre, devido a importância que esses sistemas adquiriram ao longo do tempo de acordo com a evolução dos componentes de hardware e software associados a estes. Como consequência, erros tendem a se tornar presente nesses sistemas, devido a problemas no projeto ou na implementação de falhas no hardware/software desenvolvido, mesmo que a detecção de falhas e a proteção do hardware possam ter sido analisados e modelados de forma rigorosa. O nível de robustez e complexidade desses sistemas também têm aumentado, exigindo processadores mais eficientes, memórias com maiores capacidades e interfaces de usuário que garantem ao usuário destes sistemas uma boa usabilidade e confiabilidade, aspecto que possui grande importância no processo de desenvolvimento destes sistemas. A partir desse contexto, o *framework* multiplataforma Qt surge como forte e eficaz ferramenta, qual possui como foco principal a criação de interface para usuário (IU) e o desenvolvimento de aplicações para dipositivos móveis (ou não) [2]. Por isso, o problema a ser tratado por essa

dissertação está relacionado a detecção de falhas em aplicações para dispositivos móveis (ou não) que utilizam o *framework* multiplataforma Qt. Cada aplicação requer métodos de programação específicos e a natureza crítica dessas aplicações necessitam de mecanismos de proteção de falha e certificação, a fim de reduzir as falhas existentes, garantindo assim a confiabilidade destes sistemas, evitando perdas de recursos financeiros, recursos humanos, recursos logísticos e otimizando o tempo de desenvolvimento.

Cada vez mais, empresas que possuem como atividade principal o desenvolvimento de sistemas embarcados, designam seu tempo e esforços a desenvolver novas alternativas de verificação que sejam rápidas, baratas e possuam como objetivo garantir a confiabilidade dos sistemas desenvolvidos [3]. Entre as técnicas existentes, a abordagem através da verificação de modelos é considerada a de menor custo e a mais eficiente, mas nem sempre é possível utilizá-la devido as particularidades que cada sistema possui [4, 5]. Por exemplo, o verificador *ESBMC-GPU* é capaz somente de verificar códigos de GPUs que se utilizam da linguaguem CUDA, a partir de uma representação abstrata (modelo operacional) das bibliotecas associadas a ele, mas não possui suporte a qualquer outro tipo de linguaguem utilizada por outros tipos de GPUs [13, 14]. A representação abstrata (modelo operacional) utilizado em *ESBMC-GPU* têm como objetivo aproximar-se de forma semântica ao sistema real analisado [8].

1.2 Objetivos

O objetivo geral dessa dissertação é aplicar a técnica de verificação de modelos limitados para verificar aplicações que utilizam o *framework* multiplataforma Qt, usando ferramentas de verificação e compiladores C++ para validar a taxa de cobertura, o tempo de verificação e o nível de conformidade do método proposto, quando comparado com a documentação do *framework* Qt.

Os objetivos específicos são listados a seguir:

- Modularizar e extender o suporte do modelo operacional proposto que oferece estritamente o mesmo comportamento do *framework* Qt com foco na verificação de suas propriedades.
- Verificar as aplicações que utilizam o *framework* Qt contidas no conjunto de *benckmarks* com foco em duas propriedades base: acesso de memória e manipulação de dados.

- Aplicar a metodologia de verificação proposta em aplicações reais que utilizam o *framework* Qt.

1.3 Contribuições

O respectivo modelo operacional (MO) proposto, denominado QtOM, foi ampliado com o objetivo de incluir novas funcionalidades dos principais módulos do Qt, neste caso em particular, *QtGui* e *QtCore*. De fato, as principais contribuições aqui são:

- Estruturar os módulos *QtGui* e *QtCore* de acordo com a estrutura utilizada pelo *framework* Qt.
- Corrigir o erros existentes nos *containers* baseados em templates associativos e extender o suporte de QtOM para *containers* baseados em templates sequenciais.
- Avaliar o desempenho de três solucionadores SMT (Z3 [15], Yices [16] e Boolector [17]) juntamente com a abordagem proposta utilizando o conjunto de *benchmarks* (esbmc–qt) extensivamente revisado e ampliado em relação ao trabalho anterior [10].
- Integrar QtOM ao processo de verificação de programas em C++ de verificadores que se encontram no estado da arte, neste caso, DIVINE [18] e LLBMC [19]
- Fornecer suporte à verificação de duas aplicações reais que utilizam o *framework* Qt denominadas Locomaps [20] e Geomessage [21], respectivamente.
- Avaliar o modelo operacional proposto com o intuito de analisar seu nível de conformidade em relação ao *framework* Qt.

Por fim, de acordo com o conhecimento atual em verificação de software, não há outro verificador que utilize modelos operacionais e se aplique técnicas BMC para verificar programas que utilizam o *framework* Qt sobre dispositivos de eletrônica de consumo.

1.4 Trabalhos relacionados

De acordo com a atual literatura sobre verificação, não existe outro verificador disponível que seja capaz de verificar as funcionalidades do *framework* Qt. Ao contrário, as aplicações

de eletrônica de consumo que utilizam este *framework* apresentam diversas propriedades que devem ser verificadas, como estouro aritmético, segurança de ponteiros, limite de vetores e corretude no uso de *containers*. A técnica BMC aplicada em verificação de software é usada em diversos verificadores [12, 19, 22, 23] e está se tornando cada vez mais popular, principalmente, devido ao aumento de solucionadores SMT cada vez mais sofisticados, os quais são baseados em eficientes solucionadores de satisfação Booleana (do inglês, *Boolean Satisfiability* - SAT) [15]. Entretanto, também é conhecido que não existe homogeneidade em relação a abordagem de implementação e respectivas heurísticas em solucionadores SMT [12], sendo assim, os resultados obtidos através da verificação que se utiliza solucionadores SMT é fortemente afetado por esta limitação.

Ramalho *et al.* [9] apresentam o *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) como um verificador de modelos limitados para verificar programas em C++, sendo denominado ESBMC++. ESBMC++ utiliza seus modelos operacionais para oferecer suporte a verificação das características mais complexas que a linguagem oferece, como *containers* e tratamento de exceção. Vale ressaltar que os modelos operacionais usados são uma representação abstrata das bibliotecas padrões do C++, que de forma conservadora se aproximam semanticamente das bibliotecas originais. ESBMC++ se utiliza disso para codificar as condições de verificação criadas usando diferentes teorias de base apoiadas por solucionadores SMT. Entretanto, os modelos operacionais usados possuem limitações, tendo como exemplo a verificação de métodos estáticos que se encontra incompleta e ao se utilizar listas dentro de listas.

Merz, Falke e Sinz [19] apresentam o *Low-Level Bounded Model Checker* (LLBMC) como um verificador que utiliza modelos operacionais para verificar programas baseados em ANSI-C/C++. LLBMC também usa um compilador denominado *low level virtual machine* (LLVM) que possui o objetivo de converter programas ANSI-C/C++ em uma representação intermediária utilizada pelo verificador. De forma semelhante ao ESBMC++, Merz, Falke e Sinz também utilizam solucionadores SMT para analisar as condições de verificação. Contudo, diferente da abordagem aqui proposta, o LLBMC não suporta tratamento de exceção, o que acarreta numa verificação incorreta de programas reais escritos em C++, como por exemplo, programas baseados em *Standard Template Libraries* (STL). Vale ressaltar que a representação intermediária usado pelo LLVM perde algumas informações sobre a estrutura original dos respectivos programas em C++ , como por exemplo, as relações entre classes.

Barnat *et al.* apresentam o DIVINE [18] como um verificador de modelo de estado

explícito para programas ANSI-C/C++ sequenciais e multi-tarefas, o qual possui como objetivo principal verificar a segurança de propriedades de programas assíncronos e aqueles que utilizam memória compartilhada. DIVINE faz uso de um compilador denominado Clang [24] como *front-end*, a fim de converter programas C++ em uma representação intermediária utilizada por LLVM, que logo em seguida realiza a verificação sobre o *bytecode* produzido. DIVINE também possui uma implementação de ANSI-C e das bibliotecas padrões do C++, o que lhe permite verificar programas desenvolvidos com essas linguaguens. De acordo com essa abordagem, DIVINE é capaz de criar um programa que possa ser interpretado totalmente por LLVM e logo em seguida ser verificado. Contudo, a abordagem utilizada por DIVINE é custosa, pois cria uma grande quantidade de estados explícitos, os quais todos seram analisados pela ferramenta em seu processo de verificação, sendo assim, DIVINE também possui um alto tempo de verificação.

Blanc, Groce e Kroening descrevem a verificação de programas em C++ que usam *containers* STL através de abstração de predicados [25], com o uso de tipo de dados abstrato, sendo usados para realizar a verificação de STL ao invés de usar a implementação real dos componentes STL. Na verdade, os autores mostram que a corretude pode ser realizada através de modelos operacionais, provando que a partir de condições prévias sobre operações, no mesmo modelo, acarreta em condições prévias nas bibliotecas padrões, e pós-condições podem ser tão significativas quanto as condições originais. Tal abordagem é eficiente em encontrar erros triviais em programas em C++, mas que necessita de uma pesquisa mais profunda para evitar erros e operações enganosas (isto é, ao envolver a modelagem de métodos internos). Vale ressaltar que, no presente trabalho, simulações do comportamento de certos métodos e funções superam o problema mencionado (ver Seção 3.2).

O *C Bounded Model Checker* (CBMC) implementa a técnica BMC para programas ANSI-C/C++ por meio de solucionadores SAT/SMT [22]. Vale ressaltar que o ESBMC foi construído a partir do CBMC, portanto, ambos verificadores possuem processos de verificação parecidos. Na verdade, o CBMC processa programas C/C++ usando a ferramenta goto-cc [26], a qual compila o código fonte da aplicação para um programa GOTO equivalente (ou seja, um grafo de fluxo de controle), a partir de um modelo compátivel ao GCC. A partir do programa GOTO criado, o CBMC cria uma árvore abstrata de sintaxe (do inglês, *Abstract Syntax Tree* - AST) que é convertida em um formato independente da linguaguem interna usada para as etapas restantes. O CBMC também utiliza duas funções recursivas $\mathscr{C}$ e $\mathscr{P}$ que registram as *restrições* (ou seja, premissas e atribuições de variáveis) e as *propriedades* (ou seja, condições de segu-

rança e premissas definidas pelo o usuário), respectivamente. Este verificador cria de forma automática as condições de segurança que verificam o estouro aritmético, violação no limite dos vetores e checagem de ponteiro nulo [27]. Por fim, um gerador de condições de verificação (do inglês, *Verification Condition Geneator* - VCG) cria condições de verificação (do inglês, Verification Conditions - VCs) a partir das fórmulas criadas e os envia para um solucionador SAT/SMT. Embora o CBMC esteja relacionado como susposto verificador de programas em C++, Ramalho *et al.* [9] e Merz *et al.* [28] relatam que o CBMC falha ao verificar diversas aplicações simples em C++, o que também foi confirmado neste trabalho (ver Seção 4.3).

Monteiro *et al.* [10] descreve um modelo operacional que integrado a ESBMC++ consegue verificar uma aplicação que utiliza o *framework* Qt. O modelo operacional descrito se encontra em estado inicial e não moduralizado por tipos e classes mas oferece um suporte a containers sequenciais. Vale ressaltar que no presente trabalho, o modelo operacional descrito é extendido, modularizado e são corrigidos alguns erros existentes a nível de modelo operacional (*i.e*, *namespace* implementado em classe inapropiada e a criação de rotinas de verificação em métodos estáticos). Os módulos *QtCore* e *QtGui* continuam presentes como módulos principais mas também foram adicionados outros três módulos (*QtDeclarative*, *QtWidgets*, *QtNetwork*) com o objetivo de aumentar o suporte do modelo operacional, possibilitando assim a sua aplicação na análise de aplicações reais. Internamente foram criadas representações de todas classes pertencentes a esses módulos com um total de 261 representações, assim como um suporte específico e completo a todas as classes *container* do *framework* Qt que também são amplamente utilizadas em aplicações reais

Finalmente, destaca-se que QtOM foi completamente escrito na linguagem de programação C++, o que facilita a integração dentro de processos de verificação de outros verificadores, logo, QtOM não foi somente intregado ao processo de verificação de ESBMC++ mas também ao de DIVINE [18] e LLBMC [19], afim de obter uma avaliação sobre a abordagem proposta.

1.5 Organização

Neste capítulo, inicialmente descreveram-se sobre o contexto que envolve o trabalho, a motivação, seus objetivos, além de terem sido apresentados trabalhos relacionados de acordo com a abordagem proposta, com o intuito de descrever referências sobre o tema proposto. Este

trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta uma breve introdução sobre a arquitetura do ESBMC++ e as teorias do módulo da satisfatibilidade (SMT), além de descrever um resumo sobre o *framework* multiplataforma Qt;

- O Capítulo 3 descreve uma representação simplificada das bibliotecas Qt, nomeado como Qt *Operational Model* (QtOM), que também aborda pré e pós-condições e a implementação formal de Qt *Containers* associativos e sequencias desenvolvidos de forma detalhada.

- O Capítulo 4 descreve o teste de conformidade e os resultados experimentais realizados usando *benchmarks* Qt/C++ e também a verificação de duas aplicações baseadas em Qt, onde a primeira apresenta imagens de satélites, terrenos, mapas de ruas e *Tiled Map Service* (TMS) *panning* entre outras características [20] e a segunda aplicação cria um *broadcast User Datagram Protocol* (UDP) baseado em arquivos XML.

- Por fim, o Capítulo 5 apresenta as conclusões, destacando a importância da criação de um modelo para verificar aplicações que utilizam o *framework* Qt, assim como, os trabalhos futuros também são descritos.

Capítulo 2

Fundamentação Teórica

Este capítulo descreve a arquitetura do ESBMC++ e algumas características estruturais do *framework* multiplataforma Qt. Este trabalho consiste na verificação de programas escritos em C++ baseados no *framework* multiplataforma Qt, usando a ferramenta ESBMC++, a qual possui um *front-end* baseado no CBMC com o intuito de produzir VCs para um programa Qt/C++. No entanto, em vez de passar tais VCs para um solucionador SAT, o ESBMC++ os codifica por meio de diferentes teorias de base do SMT e em seguida, passa as fórmulas associadas para um solucinador SMT.

2.1 ESBMC++

O ESBMC++ é um *context-bounded model checker* baseado em solucionadores SMT para verificar programas ANSI-C/C++ [9, 11, 12, 29]. A Figura 2.1 apresenta a arquitetura do ESBMC++. Em especial, o ESBMC++ verifica programas sequencias e multi-tarefas e analisa propriedades relacionadas à estouro aritmético, divisão por zero, indíces de vetor fora do limite, segurança de ponteiros, bloqueio fatal e corrida de dados. No ESBMC++, o processo de verificação se encontra totalmente automático, isto é, todos os processos realizados são representados nas caixas cinzas de acordo com a Figura 2.1 [30, 31].

Durante o processo de verificação, primeiramente é realizado o *parser* do código fonte a ser analisado. Na verdade, o *parser* utilizado no ESBMC++ é fortemente baseado no compilador GNU C++ [9], uma vez que a abordagem permite que o ESBMC++ encontre a maioria dos erros de sintaxe já relatados pelo GCC. Programas ANSI-C/C++/Qt são convertidos em

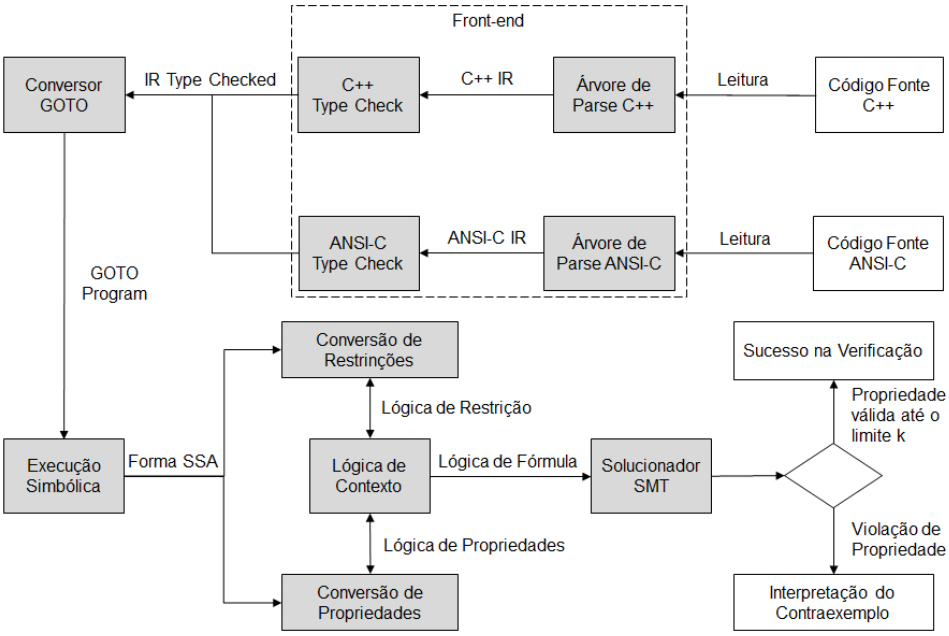


Figura 2.1: Visão geral da arquitetura do ESBMC++.

uma árvore de representação intermédiaria (do inglês, *Intermediate Representation* - IRep), e boa parte dessa representação criada é usada como base para os passos restantes da verificação. Vale ressaltar que o modelo operacional (MO) é o ponto chave neste processo de conversão, o que será explicado no Capítulo 3.

Na etapa seguinte, denominada *type-checking*, verificações adicionais são realizadas, na árvore IRep, que incluem atribuições, *type-cast*, inicialização de ponteiros e a análise das chamadas de função, assim como, a criação de *templates* e instanciações [9]. Em seguida, a árvore IRep é convertida em expressões *goto* que simplificam a representação das instruções (por exemplo, a substituição de *while* por *if* e instruções *goto*) e são executadas de forma simbólica pelo *GOTO-symex*). Como resultado, uma atribuição estática única (do inglês, *Single Static Assignment* - SSA) é criada. Baseado nisso, o ESBMC++ cria duas fórmulas chamadas *restrições* (ou seja, premissas e atribuições de variáveis) e *propriedades* (ou seja, condições de segurança e premissas definidas pelo o usuário) consideradas funções recursivas. Essas fórmulas acumulam predicados de fluxo de controle de cada ponto do programa analisado e os usa para armazernar restrições (fórmula $\mathscr{C}$) e propriedades (fórmula $\mathscr{P}$), de modo que reflita adequadamente a semântica do programa. Posteriormente, essas duas fórmulas de lógica de primeira ordem são verificadas por um solucinador SMT.

Por fim, se uma violação em alguma propriedade for encontrada, um contraexemplo é gerado pelo ESBMC++ [32], o qual atribui valores às variáveis de programa com o intuito

de reproduzir o erro encontrado. De fato, contraexemplos possuem grande importância para o diagnóstico e a análise da execução do programa, dado que as violações encontradas pode ser sistematicamente rastreadas [33].

2.2 Satisfiability Modulo Theories (SMT)

SMT determina a satisfatibilidade das fórmulas expressas em lógica de primeira ordem, usando uma combinação de suas teorias, a fim de generalizar a satisfabilidade proposicional, dando suporte às funções não interpretadas, aritmética linear e não linear, vetores de bit, tuplas, *arrays* e outras teorias de primeira ordem. Dado uma teoria $\mathscr{T}$ e uma fórmula livre de quantificadores ψ, a respectiva fórmula é satisfatível $\mathscr{T}$ se somente se existir uma estrutura que satisfaça tanto a fórmula quanto as sentenças de $\mathscr{T}$, ou seja, se $\mathscr{T} \cup \{\psi\}$ é satisfatível [34]. Dado um conjunto $\Gamma \cup \{\psi\}$ das fórmulas sobre $\mathscr{T}$, ψ é uma consequência $\mathscr{T}$ de Γ ($\Gamma \models_{\mathscr{T}} \psi$) se e somente se todo os modelos de $\mathscr{T} \cup \Gamma$ é também um modelo de ψ. Desta forma, ao verificar $\Gamma \models_{\mathscr{T}} \psi$ pode se reduzi-la para verificação de um satisfatível $\mathscr{T}$ de $\Gamma \cup \{\neg\psi\}$.

A partir do contexto descrito, as teorias de *arrays* dos solucionadores SMT são normalmente baseadas em axiomas de McCarthy [35]. A função *select(a,i)* indica o valor de *a* no índice *i* e a função *store(a,i,v)* indica um vetor que é exatamente o mesmo que *a*, a menos que o valor do índice *i* seja *v*. Formalmente, *select* e *store* pode então ser caracterizados por axiomas [15, 17, 36]

$$i = j \rightarrow select\left(store\left(a,i,v\right),j\right) = v$$

e

$$i \neq j \rightarrow select\left(store\left(a,i,v\right) = select\left(a,j\right).\right.$$

Tuplas são utilizadas para modelar *union* e *struct* em ANSI-C, além de fornecer as operações de *store* e *select*, as quais são semelhantes as usadas em vetores. No entanto, elas trabalham com elementos de tupla, isto é, cada campo de uma tupla é representado por uma constante inteira. Desta forma, a expressão $select(t,f)$ indica o campo f de uma tupla t, enquanto a expressão $store(t,f,v)$ indica uma tupla t que, no campo f, tem o valor v. A fim de analisar a

satisfabilidade de uma determinada fórmula, solucionadores SMT lidam com termos baseados em suas teorias usando um procedimento de decisão [37].

2.3 O *Framework* Multiplataforma Qt

Diversos módulos de software, conhecidos como *frameworks*, têm sido utilizados para acelerar o processo de desenvolvimento de aplicações. Diante desse contexto, o *framework* multiplataforma Qt [2] representa um bom exemplo de um conjunto de classes reutilizavéis, onde a engenharia de software presente é capaz de favorecer o desenvolvimento de aplicações gráficas que utilizam C++ [2] e Java [38]. São fornecidos programas que são executados em diferentes plataformas tanto de hardware quanto de software com o mínimo de mudanças nas aplicações desenvolvidas com o objetivo de manter o mesmo desempenho. Samsung [39], Philips [40] e Panasonic [41] são algumas das empresas presentes na lista top 10 da Fortune 500 que utilizam Qt para o desenvolvimento de suas aplicações [2].

De acordo com o relatório do *Cross-Platform Tool Benchmarking* 2014 [42], Qt é o *framework* multiplataforma que lidera o desenvolvimento de aplicações para dispositivos e interfaces para usuários. Com as suas bibliotecas organizadas em módulos, conforme mostrado na Figura 2.2, o módulo *QtCore* [2] é considerado um módulo base de Qt pois, contém todas as classes não gráficas das classes *core* e em particular contém um conjunto de bibliotecas denominado de classes *Containers* que possui como implementação um modelo base para esses tipos de classe, com um intuito de uso geral ou como uma alternativa para *containers* STL. Esses tipos de estruturas são amplamente conhecidos e usados em aplicações reais com Qt e consistem em um item muito importante nos processos de verificação.

Além desses submódulos, o *QtCore* também contêm um sistema de eventos denominado Qt *Event*, onde, Qt representa um evento por meio de um objeto que herda a classe *QEvent* (classe base para o sistema Qt *Event*) na qual contêm informações necessárias sobre todas as ações (internas ou externas) relacionadas a uma dada aplicação. Uma vez instanciado, este objeto é enviado para uma instância da classe *QObject*, o qual possui como função chamar um método escalonador apropriado para o seu tipo.

Desta forma, o *framework* multiplataforma Qt fornece uma completa abstração para aplicações que envolvem interface gráfica com o usuário (do inglês, *Graphical User Interface* - GUI) usando APIs nativas, a partir das diferentes plataformas operacionais disponíveis no

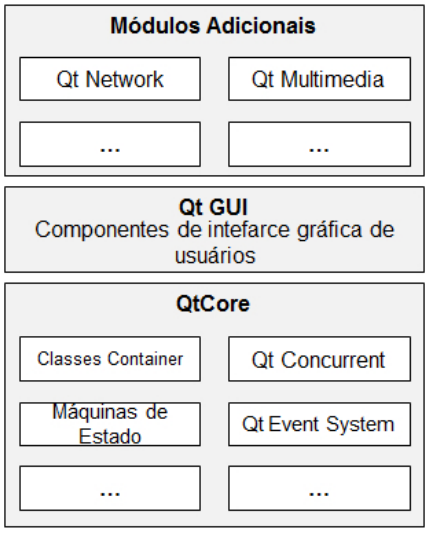


Figura 2.2: Visão geral da estrutura do *framework* multiplataforma Qt.

mercado, para consultar as métricas e desenhar os elementos gráficos. Também são oferecidos *signals* e *slots* com o objetivo de realizar a comunicação entres os objetos criados [43]. Outra característica importante a ser ressaltada deste *framework* é a presença de um compilador denominado *MetaObject*, o qual é responsável por interpretar os programas criados e gerar um código em C++ com meta informações [44].

Por fim e de acordo com o apresentado, nota-se que a complexidade e robustez de programas que utilizam o *framework* multiplataforma Qt afeta diretamente os processos de verificação relacionados a eles. Em resumo, o QtOM possui uma representação de todas as classes acima referidas e suas respectivas interações a fim de suportar também todo o sistema Qt *Event*.

2.4 Resumo

Neste capítulo, foi descrito a arquitetura de ESBMC++ sendo explicado o objetivo e a tarefa de cada etapa de seu processo de verificação de acordo com a Figura 2.1, descreveu-se também como as técnicas SMT são usadas para determinar a satisfatibilidade das fórmulas expressas em lógica de primeira ordem e por fim, foi descrito algumas características estruturais do *framework* multiplataforma Qt que de acordo com a Figura 2.2, mostra as suas bibliotecas organizadas em módulos, dentre as quais destaca-se o módulo *QtCore* [2] como um módulo base onde, estão as classes não gráficas e contém um conjunto de bibliotecas denominado de classes *Containers*. Essas estruturas são amplamentes conhecidas e usadas em aplicações reais com

Qt e consistem em um item muito importante nos processos de verificação. O módulo *QtGui* também é descrito pois, fornece uma completa abstração para aplicações que envolvem interface gráfica usando APIs nativas, outra característica importante a ser ressaltada é a presença do compilador chamado *MetaObject* [44]. Como resultado, o conteúdo apresentado nesse capítulo fornece todo o embasamento necessário para compreensão da ferramenta, técnica SMT e o *framework* utilizado neste trabalho.

Capítulo 3

Método proposto

Este capítulo descreve todo o processo de verificação de ESBMC++ junto com QtOM, cujo torna possível a verificação de aplicações escritas em C++ que utilizam o *framework* Qt, e o processo de criação de QtOM, enfatizando as classes *containers* do *framework* Qt. Inicialmente, na Seção 3.1 é descrito como QtOM auxilia ESBMC++ a transformar o código de entrada em uma árvore IRep e a identificar cada estrutura presente na aplicação e como está disposto as classes containers no *framework* Qt, assim como, o seu uso em aplicações. Logo em seguida, nas Seções 3.2 e 3.3 é descrito como QtOM se utiliza de pré e pós-condições para se obter um comportamento igual ao descrito na documentação do *framework* Qt [2]. Na Seção 3.4 é descrito a linguagem que foi utilizada como base para se criar as representações contidas em QtOM com o objetivo de formalizar a implementação de cada *container*. Por fim, nas Seções 3.5 e 3.6 é descrito como as classes containers foram divididas entre sequenciais e associativas, e o seu comportam no momento de sua utilização por QtOM através de lógica formal.

3.1 Introdução

Inicialmente, o *parser* utilizado já havia sido mencionado na Seção 2.1 do Capítulo 2 e, nesta etapa, é onde o ESBMC++ transforma o código de entrada em uma árvore IRep que contém todas as informações necessárias para o seu processo de verificação. Sendo assim, ao fim desta etapa, ESBMC++ deve identifica corretamente cada estrutura presente na aplicação analisada. No entanto, ESBMC++ apenas suporta a verificação de aplicações ANSI-C/C++.

Porém, apesar das aplicações utilizarem C++ o *framework* Qt possui suas bibliotecas nativas, as quais possuem estruturas hierárquicas próprias que inviabilizam a utilização da ferramenta. Como consequência, o processo de verificação para essas bibliotecas e suas respectivas implementações otimizadas afetariam de maneira desnecessária as VCs, além do mais poderiam não conter qualquer afirmação a respeito de propriedades específicas do *framework*, tornando a verificação uma tarefa inviável.

O uso do QtOM neste capítulo possui como objetivo solucionar o problema descrito acima, por ser uma representação simplificada que considera a estrutura de cada biblioteca e suas respectivas classes associadas, incluindo atributos, assinaturas de métodos, protótipos de funções e assertivas que garantem que cada propriedade seja formalmente verificada. Na verdade, existem muitas propriedades a serem verificadas como acesso de memória inválida, valores negativos que representam períodos de tempo, acesso a arquivos inexistentes e ponteiros nulos. Além disso, existem pré e pós-condições que são necessárias para executar corretamente os métodos do Qt. Vale ressaltar que QtOM é ligado de forma manual ao ESBMC++, logo no início do processo de verificação, como mostrado na Figura 3.1.

Desta forma, o QtOM pode auxiliar o processo de *parser* para criar uma representação intermediária em C++ que de acordo com a documentação do *framework* analisado, possui todas as assertivas indispensáveis para a verificação das propriedades mencionadas acima. A documentação do *framework* Qt é utilizada para facilitar a identificação de propriedades ou na extração de estruturas pertinentes ao *framework*. Por fim, o fluxo de verificação segue de maneira tradicional.

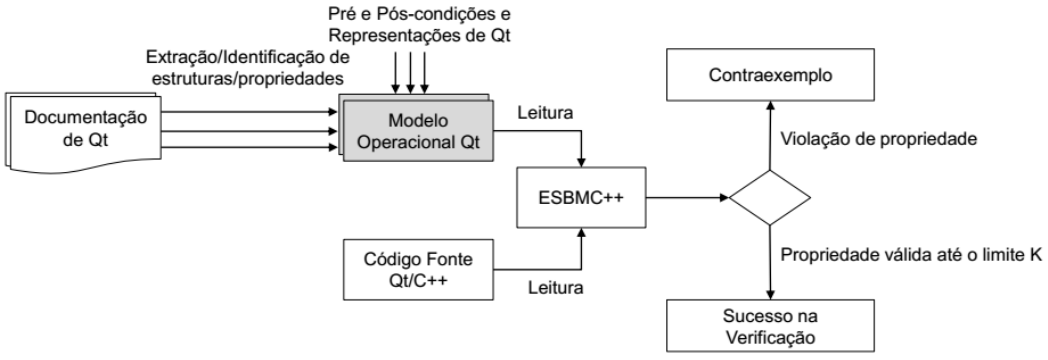


Figura 3.1: Conectando QtOM a ESBMC++.

O módulo *Qt Core* possui um subconjunto de classes denominado *Qt Container* [2] como alternativa para os *containers* da *Standard Template Library* (STL) disponíveis na lin-

guagem C++ [46]. Por exemplo, durante o desenvolvimento de uma determinada aplicação, é necessário a criação de uma pilha de tamanho variável para o armazenamento de objetos do tipo *QWidgets*, neste caso, uma alternativa seria o uso da classe *QStack* que implementa um *container* com a política LIFO (do inglês, *last-in, first-out*) (*e.g.*, *QStack*<*QWidget*>). Além disso, tais *containers* também fazem uso de estruturas denominadas *iterators* no estilo Java/STL, de modo a se deslocar ao longo dos dados armazenados no *container* criado.

Desta forma, esse subconjunto de classes pode ser classificado em dois subgrupos: sequenciais e associativos, dependendo da estrutura de armazenamento implementada. Neste contexto, as classes *QList*, *QLinkedList*, *QVector*, *QStack* e *QQueue* são classificadas como *containers* sequenciais, enquanto as classes *QMap*, *QMultiMap*, *QHash*, *QMultiHash* e *QSet* são classificadas como *containers* associativos.

3.2 Pré-condições

Ao se utilizar modelos limitados é possível diminuir a complexidade que é associada as árvores IRep. Desta forma, o ESBMC++ é capaz de construir árvores IRep de uma forma muito rápida, com baixa complexidade, mas que englobam todas as propriedades necessárias para a verificação dos programas desejados. Além disso, o uso de assertivas se torna indispensável para verificar propriedades relacionadas aos métodos do *framework* multiplataforma Qt, assim como, as suas respectivas execuções que não são contempladas pelas bibliotecas padrões. Deste modo, tais assertivas são integrados aos respectivos métodos com o objetivo de detectar violações em relação ao uso incorreto do framework Qt.

Em resumo, baseado na adição de assertivas, ESBMC++ é capaz de verificar propriedades específicas contidas em QtOM e identificar como pré-condições partes das propriedades relacionadas, ou seja, as propriedades que devem ser mantidas de forma que haja uma execução correta de um determinado método ou função. Por exemplo, a abordagem proposta pode verificar se um parâmetro, que representa uma determinada posição dentro de uma vetor é maior ou igual a zero.

De acordo com o fragmento de código mostrado na Figura 3.2 que pertence a aplicação chamada *GeoMessage Simulator*. Esta aplicação fornece mensagens para componentes do sistema na plataforma ArcGIS [21]. Como mostrado, o método *setFileName*() manipula uma pré-condição (veja linha 6), onde *m_inputFile* é um objeto da classe *QFile* que proporciona

uma interface de leitura e escrita para se manipular arquivos [2]. Ao ser definido um nome ao arquivo, o objeto *fileName* da *QString* não pode ser nulo. Desta forma, quando *setFileName*() é chamado, o ESBMC++ interpreta o seu comportamento de acordo com o implementado em QtOM como mostrado na Figura 3.3.

```
1 QString loadSimulationFile( const QString &fileName )
2 {
3     if(m_inputFile.isOpen())
4       m_inputFile.close();
5
6     m_inputFile.setFileName(fileName);
7
8     //Check file for at least one message
9     if( !doInitialRead() )
10    {
11        return QString(m_inputFile.fileName() + "is an empty message
              file");
12    }
13    else
14    {
15        return NULL;
16    }
17 }
```

Figura 3.2: Fragmento de código da função *loadSimulationFile* presente no benchmark Geomessage Simulator.

A Figura 3.3 mostra um trecho do modelo operacional correspondente à classe QFile com uma implementação de *setFileName*() (veja as linhas 5-10), onde apenas as pré-condições são verificadas.

```
1 class QFile {
2 ...
3     QFile( const QString &name ) { ... }
4     ...
5     void setFileName( const QString &name ){
6         __ESBMC_assert( !name.isEmpty(), "The string must not be
              empty" );
7         __ESBMC_assert( !this->isOpen(), "The file must be closed" );
8     }
9     ...
10 };
```

Figura 3.3: Modelo operacional de *setFileName*() presente na classe QFile.

Em particular, se o objeto QString passado como um parâmetro não estiver vazio (veja a linha 6), a operação é válida e, consequentemente, a assertiva é considerada *true*. Caso

contrário, se uma operação incorreta for realizada, como uma *string* vazia for passada como parâmetro, a assertiva é considerada *false*. Nesse caso, o ESBMC++ retornaria um contraexemplo com todos os passos necessários para reproduzir a execução de tal violação, além de descrever o erro da respectiva premissa violada.

Do ponto de vista da verificação de software, existem métodos/funções que também não apresentam qualquer propriedade a ser verificada como aqueles cuja única finalidade é imprimir valores na tela. Dado que ESBMC++ realiza o processo verificação a nível de software ao invés de testar o hardware, a corretude a cerca do valor impresso na tela não foi abordado neste trabalho. Por fim, para tais métodos só existem assinaturas, de modo que o verificador proposto seja capaz de reconhecer a estrutura desejada para que durante o processo de análise seja construído uma árvores IRep confiável. No entanto, eles não apresentam qualquer modelagem (corpo de função), uma vez que não exista nenhuma propriedade a nível de software a ser verificada.

3.3 Pós-condições

Em aplicações reais existem métodos que não contêm apenas propriedades que serão tratadas como pré-condição, mas também serão considerados como pós-condições [10, 45]. Por exemplo, de acordo com a documentação do *framework* Qt [2], a função $setFileName$ conforme descrita na Seção 3.2 não deve ser utilizada se o respectivo arquivo passado como parâmetro já estiver sendo utilizado, o que é verificado em seu código fonte a partir da estrutura if presente nas linhas 3 e 4 visto na Figura 3.2.

No entanto, a execução da instrução presente na linha 4 (veja Fig. 3.2) seria não determinística para o ESBMC++, assim como a assertiva presente na linha 7 (veja Fig. 3.3) do modelo operacional associado, uma vez que não há como se afirmar se o respectivo arquivo está sendo utilizado ou não. Desta forma, é evidente que será necessário simular o comportamento do método *isOpen*() com o intuito de verificar de forma coerente as propriedades relacionadas com manipulação de arquivos. Como a classe *QFile* indiretamente herda, a partir da classe *QIODevice*, os métodos *open*() e *isOpen*(), se cria um modelo operacional para QIODevice conforme mostrado na Figura 3.4 a partir do descrito na documentação do *framework* com suas representações e simulações comportamentais.

```
1  class QIODevice {
2      ...
3      bool QIODevice::open(OpenMode mode){
4          this->__openMode = mode;
5          if (this->__openMode == NotOpen)
6              this->__isOpen = false;
7          this->__isOpen = true;
8      }
9
10     bool isOpen() const{
11         return this->__isOpen;
12     }
13     ...
14 private:
15     bool __isOpen;
16     OpenMode __openMode;
17     ...
18 };
```

Figura 3.4: Modelo operacional para os métodos *open*() e *isOpen*() na classe *QIODevice*.

A Figura 3.5 demonstra como foi criado o modelo operacional para a classe *QIODevice* descrito acima em QtOM a partir da documentação do *framework* Qt.

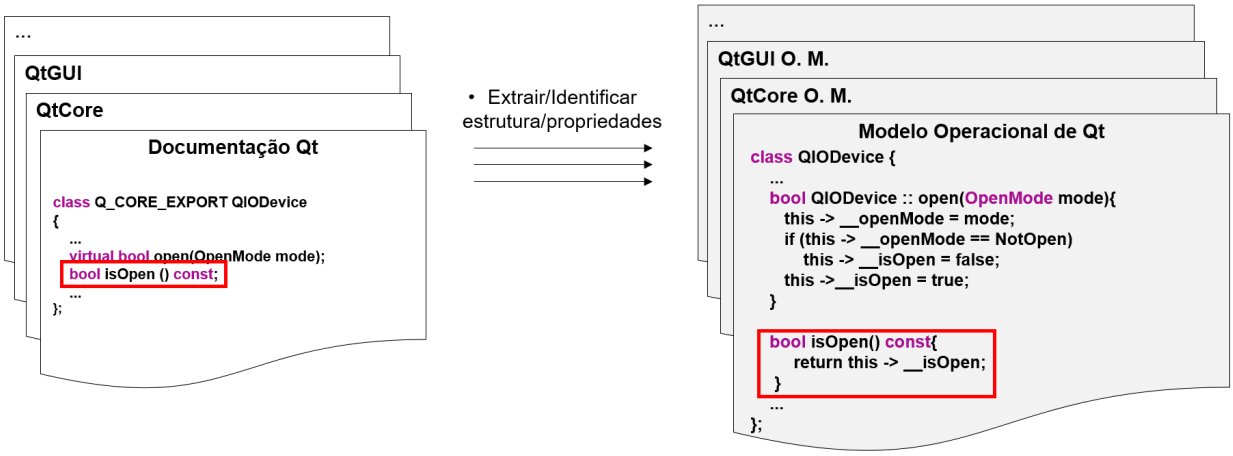


Figura 3.5: Modelo operacional para os métodos *open*() e *isOpen*() na classe *QIODevice*.

Por fim, todos os métodos que resultam em uma condição que deve ser mantida, a fim de permitir uma adequada execução de futuras instruções, devem apresentar uma simulação do seu comportamento. Consequentemente, um determinado modelo operacional deve seguir de forma rigorosa as especificações descritas na documentação oficial do *framework* [2] com o objetivo de se obter o mesmo comportamento, assim como, incluir mecanismos necessários a verificação do código. Como resultado, é necessário verificar o nível de equivalência entre o modelo operacional e a biblioteca original, com o intuito de se comparar o comportamento de ambos, tendo

em vista que os modelos operacionais desenvolvidos são uma cópia simplificada das bibliotecas originais com todos os mecanismos necessários para a verificação do código [13].

3.4 Linguagem Base

Com o intuito de implementar o modelo operacional para as classes que compõe o *Qt Container*, a formalização da linguagem base descrita por Ramalho *et al.* [9] é utilizada e se estende até a formulação das propriedades $\mathscr{C}$ e $\mathscr{P}$. Contudo, tal linguagem foi adaptada, neste trabalho, com o objetivo de formular adequadamente a verificação de ambos tipos de *containers* (*i.e.*, sequencial ou associativo) como mostrado na Figura 3.6.

De acordo com a Figura 3.6, os elementos básicos estão divididos em dois domínios sintáticos: V para valores e K para as chaves.

$$
\begin{aligned}
V &::= v \mid I_v \mid P_v \\
K &::= k \mid I_k \mid P_k \\
I &::= i \mid C.begin() \mid C.end() \\
&\quad \mid C.insert(I, V, \mathbb{N}) \mid C.erase(I) \mid C.search(V) \\
&\quad \mid C.insert(K, V) \mid C.search(K) \\
P &::= p \mid P(+ \mid -)P \mid C_k \mid C_v \mid I_k \mid I_v \\
C &::= c \\
\mathbb{N} &::= n \mid \mathbb{N}(+ \mid * \mid \ldots)\mathbb{N} \mid I_{pos} \mid C_{size}
\end{aligned}
$$

Figura 3.6: Sintaxe da linguagem adaptada, base para a descrição formal dos *containers*.

No entanto, os demais domínios, I, P, $\mathbb{N}$ e C são mantidos por *iterators*, ponteiros, índices inteiros e expressões *container* adequadas, respectivamente. Assim, as variáveis k do tipo K e v do tipo V são adicionadas. Dessa forma, a notação I_v representa um valor armazenado em um *container* em uma posição direcionada pelo *iterator* I e I_k representa uma chave armazenada em um *container* em uma posição direcionada também pelo *iterator* I. Tais notações são abreviações para $store(i, I_{pos}, I_v)$ e $store(i, I_{pos}, I_k)$, respectivamente, onde a expressão $store(t, f, v)$ indica *container* t que no campo f possui um valor v. Da forma similar, P_k e P_v representam ponteiros para a chave e o valor, respectivamente.

Além disso, três outros métodos foram incluídos com o objetivo de descrever as operações realizadas em cada *container*. O método $C.insert(k, v)$ insere um valor v no *container* C com uma chave correspondente k e possui como retorno um *iterator* que aponta para o novo

elemento inserido. O método $C.search(k)$ retorna um *iterator* que aponta para a primeira evidência de um elemento com uma chave k correspondente. De modo semelhante, $C.search(v)$ também retorna um *iterator* que aponta para a primeira evidência de um elemento com um valor v correspondente. No entanto, caso não exista nenhuma chave ou valor correspondente durante a operação com tais métodos, os mesmos retornarão $C.end()$, o qual corresponde ao *iterator* que aponta para a posição imediatamente posterior ao último elemento. Por fim, C_k é um endereço de memória que armazena o início das chaves dos *containers*, assim como, C_v é usado para armazenar os valores dos *containers*.

É importante ressaltar que, todas demais operações provenientes da linguagem base mencionada são utilizadas aqui de acordo como descrito por Ramalho *et al.* [9].

3.5 *Containers* Sequenciais

Containers sequenciais tem como objetivo armazenar elementos em uma determinada ordem [47]. De acordo com a documentação do Qt [2], *QList* é a classe *container* mais utilizada e possui uma estrutura em formato de lista encadeada. Da mesma forma, *QLinkedList* também possui um estrutura em forma de lista, embora seja acessada através de *iterators* ao invés de índices inteiros. Na classe *QVector*, há presente uma estrutura de *array* expansível e, por fim, *QStack* e *QQueue* fornecem estruturas que implementam diretivas como LIFO e FIFO (em inglês, *first-in, first-out*), respectivamente.

Para simular adequadamente os *containers* sequenciais, os modelos propostos utilizam da linguagem base descrita na Seção 3.4. Os *containers* sequenciais são implementados a partir de um ponteiro C_v para os valores do *container* e também com um C_{size}, o qual é utilizado para representar o tamanho do respectivo *container* (onde $C_{size} \in \mathbb{N}$). Dessa forma, os *iterators* são modelados por meio de duas variáveis, uma do tipo $\mathbb{N}$ que é denominada de i_{pos} e contém o valor do índice apontado por um *iterator* e outra do tipo P que é chamado por I_v e aponta para um *container* subjacente.

Vale ressaltar que todos os métodos, a partir dessas bibliotecas, podem ser expressos em variações simplificadas de três operações principais, *insertion*($C.insert(I,V,\mathbb{N})$), *deletion*($C.erase(I)$) e *search*($C.search(V)$). A partir da transformação SSA, os efeitos adversos sobre *iterators* e *containers* são explícitos para que as operações retornem novos *iterators* e *containers*.

Por exemplo, um *container c* com uma chamada $c.search(v)$ representa uma pesquisa pelo elemento v no respectivo *container*. Deste modo, se esse elemento for encontrado, é retornado um *iterator* que aponta para o respectivo elemento, caso contrário, é retornado um *iterator* que aponta para a posição imediatamente posterior ao último elemento do *container* (*i.e.*, $c.end()$). Desta forma, a instrução "$c.search(v)$;" torna-se "$(c',i') = c.search(v)$;" que possuem efeitos adversos de forma explícita. Assim, a função de tradução $\mathscr{C}$ descreve premissas que estão relacionadas com o "antes" e o "depois" das respectivas versões das variáveis do modelo. Na verdade, notações com apóstrofe (*e.g.*, c' and i') representam o estado das variáveis do modelo, após realizar a execução da respectiva operação; notações simplificadas (*e.g.*, c and i) representam os estados anteriores. Além disso, $select(c, i = lower_{bound} \ldots i = upper_{bound})$ representa uma expressão de *loop* (*e.g.*, for e $while$), onde cada valor de c, a partir da posição $lower_{bound}$ até $upper_{bound}$, será selecionado. Da mesma forma, $store(c_1, lower^1_{bound}, select(c_2, lower^2_{bound}))$ $\ldots store(c_1, upper^1_{bound}, select(c_2, upper^2_{bound}))$ também representa uma expressão de *loop*, onde cada valor de c_2, a partir de posição $lower^2_{bound}$ até a posição $upper^2_{bound}$ serão armazenados em c_1 nas posições $lower^1_{bound}$ até $upper^1_{bound}$, respectivamente. Sendo assim,

$$
\begin{aligned}
&\mathscr{C}((c',i') = c.search(v)) := \\
&\quad \wedge i' := c.begin() \\
&\quad \wedge g_0 := \ select(c_v, i_{pos} = 0 \ldots i_{pos} = c_{size} - 1) == v \\
&\quad \wedge {i'}_{pos} := ite(g_0, i_{pos}, c_{size}) \\
&\quad \wedge {i'}_v := {c'}_v.
\end{aligned}
$$

Com relação aos *containers* sequenciais, os métodos $C.insert(I, V, \mathbb{N})$ e $C.erase(I)$ se comportam como descrito por Ramalho *et al.* [9].

3.6 *Containers* Associativos

O grupo dos *containers* associativos possui cinco classes: *QMap*, *QMultimap*, *QHash*, *QMultiHash* e *QSet*. *QMap* tem como abordagem um vetor associativo que conecta cada umas das chaves, de um certo tipo K, a um valor de um certo tipo V, onde as chaves associadas são armazenadas em ordem. Por um lado, *QHash* apresenta um comportamento similar ao *QMap*, contudo os dados armazenados possuem uma ordem arbitrária. *QMultiMap* e *QMultihash* representam respectivamente subclasses de *QMap* e *QHash*, no entanto, ambas classes permitem o

armazenamento de valores replicados. Por fim, *QSet* armazena objetos que estão associados a um conjunto de valores ordenados.

Com o intuito de implementar *containers* associativos, um ponteiro c_v é definido para os valores armazenados, um ponteiro c_k é utilizado para armazenar as chaves do respectivo *container* e uma variável c_{size} é utilizada para guardar a quantidade de elementos inseridos. Em especial, c_k e c_v estão conectados por meio de um índice, ou seja, dado um *container* c que contém uma chave k e um valor v assume-se que

$$[\forall \omega \in \mathbb{N} | 0 \leq \omega < c_{size}]$$

e

$$k \rightarrow v \iff select\,(c_k, \omega) = k \wedge select\,(c_v, \omega) = v,$$

onde $(k \rightarrow v)$ indica que uma chave k é associada a um valor v and ω representa uma posição válida em c_k e c_v. Além disso, a função *select(a, i)* indica o valor de a em um índice i [9]. Novamente, todas as operações dessas bibliotecas podem ser expressadas a partir de uma variação simplificada das três principais operações citadas na Seção 3.5.

Portanto, a operação de inserção para *containers* associativos pode ser realizada de duas maneiras diferentes. Em primeiro lugar, se a ordem não importa, um novo elemento é inserido no final de c_k e c_v. Desta forma, dado um container c, o método $c.insert(k, v)$ ao ser chamado realiza inserções de elementos no *container* c com um o valor v e associado a uma chave k, porém se k já existe, ele substitui o valor associado a k por v e retorna um *iterator* que aponta

para o elemento inserido ou modificado. Deste modo,

$$
\begin{aligned}
&\mathscr{C}((c', i') = c.insert(k, v)) := \\
&\quad \wedge\, c'_{size} := c_{size} + 1 \\
&\quad \wedge\, i' := c.begin() \\
&\quad \wedge\, g_0 := \ select(c_k, i_{pos} = 0 \ldots i_{pos} = c_{size} - 1) == k \\
&\quad \wedge\, i'_{pos} := ite(g_0, i_{pos}, c_{size}) \\
&\quad \wedge\, c'_k := \ store(c_k, i'_{pos} + 1, select(c_k, i'_{pos})), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad store(c_k, c_{size}, select(c_k, c_{size} - 1))) \\
&\quad \wedge\, c'_v := \ store(c_v, i'_{pos} + 1, select(c_v, i'_{pos})), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad store(c_v, c_{size}, select(c_v, c_{size} - 1))) \\
&\quad \wedge\, c'_k := store(c_k, i'_{pos}, k) \\
&\quad \wedge\, c'_v := store(c_v, i'_{pos}, v) \\
&\quad \wedge\, i'_k := c'_k \\
&\quad \wedge\, i'_v := c'_v.
\end{aligned}
$$

Em uma outra versão do método de inserção onde a ordem das chaves possuem importância. Todas as variáveis citadas acima são consideradas e uma comparação é realizada, a fim de assegurar que o novo elemento é inserido na ordem desejada. Assim,

$$
\begin{aligned}
&\mathscr{C}((c', i') = c.insert(k, v)) := \\
&\quad \wedge\, c'_{size} := c_{size} + 1 \\
&\quad \wedge\, i' := c.begin() \\
&\quad \wedge\, g_0 := \ select(c_k, i_{pos} = 0 \ldots i_{pos} = c_{size} - 1) > k \\
&\quad \wedge\, g_1 := \ select(c_k, i_{pos} = 0 \ldots i_{pos} = c_{size} - 1) == k \\
&\quad \wedge\, i'_{pos} := ite(g_0 \vee g_1, i_{pos}, c_{size}) \\
&\quad \wedge\, c'_k := \ store(c_k, i'_{pos} + 1, select(c_k, i'_{pos})), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad store(c_k, c_{size}, select(c_k, c_{size} - 1))) \\
&\quad \wedge\, c'_v := \ store(c_v, i'_{pos} + 1, select(c_v, i'_{pos})), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad store(c_v, c_{size}, select(c_v, c_{size} - 1))) \\
&\quad \wedge\, c'_k := store(c_k, i'_{pos}, k) \\
&\quad \wedge\, c'_v := store(c_v, i'_{pos}, v) \\
&\quad \wedge\, i'_k := c'_k \\
&\quad \wedge\, i'_v := c'_v.
\end{aligned}
$$

Em casos onde chaves com vários valores associados são permitidos, a comparação feita será ignorada, caso seja verificado que já existe um elemento associado a respectiva chave analisada. Por fim, com o propósito de realizar uma exclusão, o método apagar, o qual é representado por *erase*(*i*) onde *i* é um *iterator* que aponta para o elemento a ser excluído. Isso exclui o elemento apontado por *i*, movendo para trás todos os elementos seguidos pelo elemento que

foi excluído. Deste modo,

$$
\begin{aligned}
&\mathscr{C}((c',i') = c.erase(i)) := \\
&\quad \wedge\, c'_{size} := c_{size} - 1 \\
&\quad \wedge\, c'_k := \; store(c_k, i'_{pos}, select(c_k, i'_{pos} + 1)), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad store(c_k, c_{size} - 2, select(c_k, c_{size} - 1))) \\
&\quad \wedge\, c'_v := \; store(c_v, i'_{pos}, select(c_v, i'_{pos} + 1)), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad store(c_v, c_{size} - 2, select(c_v, c_{size} - 1))) \\
&\quad \wedge\, i'_k := c'_k \\
&\quad \wedge\, i'_v := c'_v \\
&\quad \wedge\, i'_{pos} := i_{pos} + 1.
\end{aligned}
$$

Nota-se que tais modelos induzem implicitamente duas propriedades principais que possuem o objetivo de executar de forma correta as operações já mencionadas. A princípio a primeira propriedade se torna evidente quando c_k e c_v são considerados não vazios, isto é, c_{size} também não é nulo para as operações de busca e exclusão de elementos. A segunda propriedade se torna evidente quando i é considerado um *iterator* do respectivo *container* referido, isto é, dado um *container* c com os ponteiros bases c_k e c_v, $i_k = c_k$ e $i_v = c_v$ são mantidos.

3.7 Resumo

Neste capítulo, foi descrito como é realizado o processo de verificação de ESBMC++ junto com o modelo operacional (QtOM) de acordo como mostrado na Figura 3.1 e como subconjunto denominado *Qt Container* que está integrado ao módulo *Qt Core* do *framework* Qt, sendo classificado em dois subgrupos: sequenciais e associativos, dependendo da estrutura de armazenamento implementada. Logo em seguida, é descrito como QtOM se utiliza de pré-condições para identificar partes das propriedades relacionadas ao *framework* multiplataforma Qt, para que um determinado método ou função seja executado de forma correta e como QtOM usa pós-condições para garantir o mesmo comportamento de métodos ou funções de acordo com o apresentado nas bibliotecas nativas do *framework* Qt [2]. É descrito a linguaguem base formalizada por Ramalho *et al.* [9], mas adaptada com o objetivo de formular adequadamente a

verificação de ambos tipos de *containers* existentes. Por fim, os *containers* sequenciais e associativos foram descritos de forma detalhada, a fim de ressaltar todos os metódos que constituem essas bibliotecas assim como, a forma que estão implementados a partir de uma transformação SSA. Como resultado, o conteúdo apresentado nesse capítulo fornece todo o embasamento necessário para compreensão da verificação de programas Qt/C++, utilizando o ESBMC++ como ferramenta base e um modelo operacional que utiliza pré e pós-condições para analisar o *framework* em questão, enfatizando o seu subconjunto *Qt Container*.

Capítulo 4

Avaliação experimental

Este capítulo é dividido em cinco partes. A Seção 4.1 descreve toda configuração experimental utilizada, os experimentos e todos os parâmetros de avaliação utilizados. Na Seção 4.2, o desempenho e a eficácia do método proposto são analisados junto a outros solucionadores SMT (Z3, Boolector e Yices 2) utilizando programas sequenciais Qt/C++ baseados em *The Qt Documentation* [48]. Na Seção 4.3 é descrito os resultados obtidos acerca da verificação de todos os casos de teste presentes na suite de teste *esbmc–qt* utilizando ferramentas de verificação distintas (ESBMC++, LLBMC e DIVINE) com o objetivo de analisar a versatilidade e a eficácia de QtOM junto a outras abordagens de verificação. Na Seção 4.4 é descrito os resultados da verificação de duas aplicações reais (*Locomaps* [20] e *GeoMessage Simulator* [21]) utilizando QtOM. Por fim, é descrito na Seção 4.5 os resultados da comparação entre os comportamentos do *framework* Qt em relação a QtOM com o intuito de se avaliar o nível de conformidade entre ambos.

4.1 Configuração dos Experimentos

Com o intuito de avaliar a eficácia do método proposto acerca da verificação de programas que utilizam o *framework* Qt, foi criado um conjunto de testes automáticos denomidado *esbmc–qt*. Em resumo, este conjunto de testes contém 711 programas Qt/C++ (12903 linhas de código) baseados em *The Qt Documentation* [48], ou seja, todos os casos de teste utilizadas na atual avaliação.

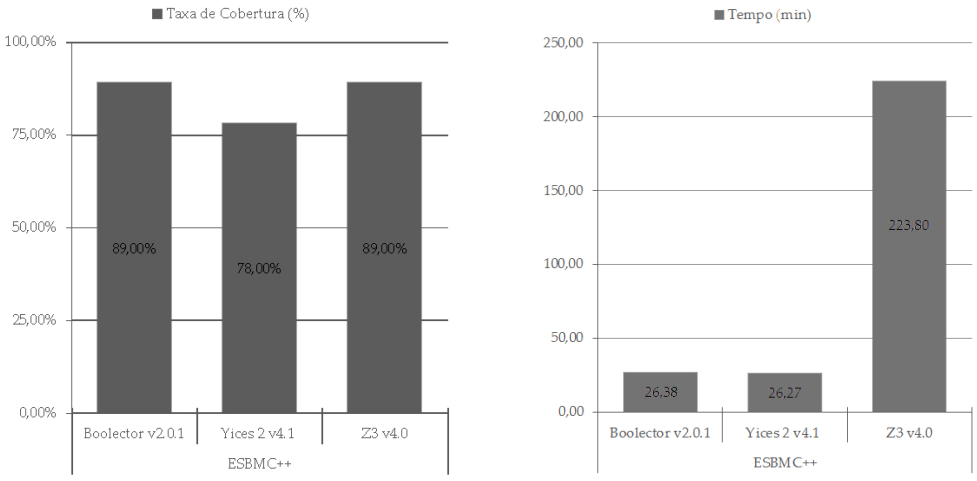
Os casos de teste mencionados acima estão divididos em 10 principais conjuntos de teste,

denominados QHash, QLinkedList, QList, QMap, QMultiHash, QMultiMap, QQueue, QSet, QStack e QVector. Vale ressaltar que todos os conjuntos de teste criados possuem casos de teste de acordo com a respectiva classe *container* a ser analisada que em sua maioria possuem acesso aos módulos *Qt Core* e *Qt GUI* (e.g, conjunto de teste QHash possui casos de testes que foram utilizados para avaliar o processo de verificação da classe *container QHash* pertencente ao *framework* multiplataforma Qt). Os casos de teste foram desenvolvidos a partir da documentação referente ao *framework* Qt [48] e como o objetivo de analisar todas as característica fornecidas pelo *framework*. Além disso, cada caso de teste é verificado manualmente antes de ser adicionado ao seu respectivo conjunto de teste. Dessa forma, é capaz de se identificar se um determinado caso de teste possui ou não qualquer erro e está de acordo com a operação a ser realizada. Além do mais, é possível garantir que 353 dos 711 casos de teste contêm erro (*i.e.*, 49.65% do casos de teste analisados contêm erro) e 358 casos de teste não possuem falhas (*i.e.*, 50.35% dos casos de teste analisados são considerados corretos). Na realidade esse tipo de inspeção sobre os casos de teste é essencial para a avaliação experimental, uma vez que pode-se comparar os resultados obtidos através da verificação realizada pelas ferramentas de verificação utilizadas e, consequentemente, avaliar adequadamente se erros reais foram encontrados.

Todos os experimentos foram realizados em um computador Intel Core *i*7-4790 com 3.60 GHz de clock e 24 GB (22 GB de memória RAM e 2 GB de mémoria virtual), utilizando-se um sistema operacional *open source* de 64 bits denominado Fedora. Além disso, também se utilizou a ferramenta de verificação ESBMC++ *v*1.25.4 com três tipos de solucionadores SMT instalados (Z3 *v*4.0, Boolector *v*2.0.1 e Yices 2 *v*4.1). Os limites de tempo e memória utilizados para cada caso de teste foram definidos em 600 segundos e 22 GB, respectivamente. Por fim, uma avaliação foi realizada utilizando o CBMC *v*5.1, LLBMC *v*2013.1 e DIVINE *v*3.3.2, combinados com o modelo operacional (QtOM), com o objetivo de proporcionar comparações entre ferramentas com diferentes abordagens de verificação. Os períodos de tempo foram indicados usando a função `clock_gettime`, pertencente a biblioteca `time.h` [49], que tem como finalidade aferir a hora do sistema operacional utilizado no momento de sua chamada. Para se determinar os períodos de tempo, esta função foi chamada no início e no final da aplicação assim determinando o seu tempo inicial e o seu tempo final. Logo em seguida, no final da aplicação é determinado a variação entre os tempos obtidos e por fim, determinado o período de tempo em que foi executado a análise sobre a aplicação. O período de tempo é expresso em minutos (min) ou em segundos (s).

4.2 Comparação entre solucionadores SMT

É conhecido que diferentes solucionadores SMT podem afetar fortemente os resultados obtidos [12], uma vez que não existe homogeneidade em relação a abordagem de implementação e respectivas heurísticas. Primeiramente, as verificações que foram realizadas, utilizaram os três solucionadores SMT já mencionados (Z3, Boolector e Yices 2) com o objetivo de avaliar o desempenho e a eficácia do método proposto. Dessa forma, Yices 2 obteve os piores resultados, apresentando uma taxa de cobertura de 78% e um tempo de verificação de $26,27$ minutos. Além do mais, não conseguiu resolver corretamente as fórmulas SMT originadas do processo de verificação dos diversos casos de teste. Por outro lado, tanto os solucionadores Z3 quanto Boolector apresentaram uma taxa de cobertura de 89%, apesar do tempo de verificação obtido ao se utilizar o solucionador Z3, para realizar verificações, seja "pior" do que ao se utilizar o solucionador Boolector, onde, respectivamente, são apresentados tempos de verificação de $223,6$ minutos e $26,38$ minutos. A partir dos resultados mencionados, o solucionador Boolector mostrou-se $8,5$ vezes mais rápido do que o solucionador Z3, apesar da mesma precisão de ambos, mas quatro casos de teste relataram violações a cerca do tempo limite determinado. Além disso, o Yices 2 não conseguiu analisar completamente todos os conjuntos de teste desenvolvidos, pois não possui suporte a tuplas e consequentemente apresentando a menor taxa de cobertura e o menor tempo. Em resumo, de acordo com a Figura 4.1, o Boolector se apresenta como o melhor solucionador para o processo de verificação proposto, pois apresenta em menor tempo a maior taxa de cobertura.



(a) Taxa de cobertura (b) Tempo de verificação

Figura 4.1: Comparação entre solucionadores SMT.

4.3 Verificação dos resultados para o *esbmc–qt*

Todos os casos de teste presentes na suíte de teste *esbmc–qt* foram verificados de forma automática por ESBMC++ com o objetivo de analisar a sua corretude e a eficácia de QtOM. Além da comparação entre solucionadores SMT descrita na Seção 4.2, uma outra análise com o objetivo de avaliar o desempenho de QtOM junto a outras abordagens de verificação também foi realizada. Como já mencionado, não existe uma ferramenta de verificação que verifique o *framework* Qt e nem um modelo operacional semelhante a QtOM utilizando a linguagem C++. No entanto, devido a versatilidade de QtOM, também é possível conectá-lo ao processo de verificação de outros verificadores de modelo como LLBMC [28] e DIVINE [18], cuja base deste processo é a tradução do código fonte em um representação intermediária denominada LLVM. Dessa forma, QtOM é usado como apoio em seus processos de tradução, pois o *bytecode* que é produzido, contém informações a cerca do código fonte utilizado na verificação e do modelo operacional (QtOM). Por fim, foi feito uma comparação em relação ao desempenho do LLBMC e ESBMC++, que são verificadores baseados em técnicas SMT, e DIVINE que emprega uma verificação de modelos através de estados explícitos. Inicialmente, houve uma tentativa de se realizar também uma comparação utilizando o verificador CBMC [22], junto com o QtOM, mas devido à falhas do verificador em relação a verificação da linguagem C++ não foi possível realizar as verificações previstas. Isto já havia sido relatado em trabalhos anteriores por Ramalho *et al.* [9] e Merz *et al.* [28], o que ocasionou em sua remoção durante o processo de avaliação.

As ferramentas utilizadas foram executadas seguindo três roteiros. Um para ESBMC++[1] que identifica a partir de um arquivo seus parâmetros iniciais e realiza sua execução, outro para LLBMC que usando CLang[2] [24] compila o código fonte desejado criando seu *bytecode* e logo em seguida, também a partir de um arquivo identifica seus parâmetros iniciais e realiza a sua execução[3] e outro para DIVINE que também pré-compila os códigos fontes em C++ criando seus respectivos *bytecode*[4] para que em seguida realize sua verificação[5] sobre eles. O desdobramento de laços é definido para cada ferramenta, isto é, o valor de <*bound*> varia entre os

[1] esbmc *.cpp --unwind <bound> --no-unwinding-assertions -I /home/libraries/ --memlimit 14000000 --timeout 600

[2] /usr/bin/clang++ -c -g -emit-llvm *.cpp -fno-exceptions

[3] llbmc *.cpp –ignore-missing-function-bodies –max-loop-iterations=<bound> –no-max-loop-iterations-checks

[4] divine compile –llvm -o main.bc *.cpp

[5] divine verify main.bc –max-time=600 –max-memory=14000 -d

casos de teste. Por enquanto, o LLBMC não suporta tratamento de exceção e os *bytecodes* que foram criados estavam com a opção *-fno-exceptions* ativa em seu compilador. Vale ressaltar, que se está opção estiver ativa, o LLBMC sempre abortará durante seu processo de verificação.

A Tabela 4.1 mostra os resultados experimentais para as junções entre QtOM e LLBMC, DIVINE e ESBMC++ usando Boolector como seu principal solucinador SMT. Vale ressaltar que o ESBMC++ utilizando Boolector não apresenta estouro de memória e tempo em qualquer caso de teste utilizado. *TC* representa o número de programas Qt/C++, *L* representa a quantidade total de linhas de código, *Time* representa o tempo total da verificação, *P* representa o número de casos de teste sem defeitos (*i.e.*, resultados positivos corretos), *N* representa o número de casos de teste com defeitos (*i.e.*, resultados negativos corretos), *FP* representa o número falsos positivos obtidos (*i.e.*, a ferramenta relata programas que estão corretos como incorretos), *FN* representa o número de falsos negativos obtidos (*i.e.*, a ferramenta relata programas incorretos como corretos) e *Fail* representa o número de erros internos obtidos durante a verificação (e.g., erros de análise).

			ESBMC++ v1.25.4						LLBMC v2013.1						DIVINE v3.3.2					
Suíte de teste	**TC**	**L**	**Time**	**P**	**N**	**FP**	**FN**	**Fail**	**Time**	**P**	**N**	**FP**	**FN**	**Fail**	**Time**	**P**	**N**	**FP**	**FN**	**Fail**
QHash	74	1170	117.2	33	33	4	4	0	37.13	31	37	0	6	0	1432.5	32	33	0	1	8
QLinkedList	87	1700	77.0	40	39	2	2	4	23.3	18	41	2	26	0	1907.6	30	42	1	14	0
QList	124	2317	102.1	53	55	7	9	0	19.4	28	56	0	28	12	2599.7	52	56	0	4	12
QMap	99	1989	277.2	42	39	10	8	0	406.4	41	46	2	8	2	2109.9	40	44	0	5	10
QMultiHash	24	363	186.4	12	12	0	0	0	30.8	12	12	0	0	0	466.3	13	12	0	0	0
QMultiMap	26	504	136.9	13	13	0	0	0	32.0	13	13	0	0	0	549.9	14	13	0	0	0
QQueue	16	299	191	8	8	0	0	0	3.9	8	8	0	0	0	339.7	8	8	0	0	0
QSet	94	1702	500.5	43	43	4	4	0	132.6	40	44	1	5	4	1897.2	40	41	0	0	13
QStack	12	280	14.5	5	5	0	0	2	2.2	6	5	1	0	0	262.1	6	6	0	0	0
QVector	152	2582	157.3	67	68	7	8	2	1825.7	44	73	0	29	6	3057.5	68	72	0	6	6
Total	708	12903	1760	316	315	34	35	8	2513.5	241	335	6	102	24	14722.4	303	327	1	30	49

Tabela 4.1: Resultados obtidos da comparação entre ESBMC++ v1.25.4 (usando Boolector como solucionador SMT), LLBMC v2013.1 e DIVINE v3.3.2.

De acordo com a Tabela 4.1, apenas $1,1\%$ dos casos de teste com o ESBMC++ alegaram falhas durante sua verificação, isto ocorre quando a ferramenta não é capaz de realizar a verificação de um determinado programa devido a erros internos encontrados. DIVINE e LLBMC apresentam taxas de falha de $6,9\%$ e $3,4\%$, respectivamente, devido não conseguirem criar os *bytecodes* de determinados casos de teste ou durante a verificação realizada foi relatado que houve um estouro de memória ou de tempo. Em relação aos resultados considerados fal-

sos positivos (*FP*), DIVINE obteve o melhor desempenho seguido por LLBMC e ESBMC++. Contudo, ESBMC++ obteve a taxa mais baixa em relação aos resultados considerados falsos negativos (*FN*) seguido por DIVINE e por fim LLBMC, isso ocorre devido a forma com que os *iterators* estão implementados em QtOM, por meio de ponteiros e vetores com o objetivo de simular seus comportamentos de forma real e de acordo com o visto no Capítulo 3. No entanto, a estrutura criada não cobre todos os comportamentos descritos na documentação do *framework* [48]. Em particular, quando uma operação de remoção de um elemento é realizada em um *container* que existe mais de um *iterator* apontando para ele, todos os *iterators* que apontam para o elemento que foi removido serão perdidos. Dessa forma, este comportamento afetará diretamente as pós-condições de um programa que consequentemente também influenciará nos resultados obtidos em relação a falsos positivos (*FP*) e falsos negativos (*FN*). Além do mais, vale ressaltar que vetores e ponteiros têm sido extensivamente utilizados de modo a obter estruturas mais simples, ou seja, sem classes e *structs* em suas representações o que, consequentemente, diminui a complexidade do processo de verificação (ver Seção 3.3). Por fim, os resultados da combinação entre ESBMC++ e QtOM gera um verificador robusto mas que ainda possui algumas lacunas a serem preenchidas a respeito do suporte a linguagem C++ como descrito por Ramalho *et al.* [9]. Vale mencionar que o nível de complexidade ao se verificar o código fonte de um programa aumenta de acordo com a quantidade de estruturas (dados, repetição, seleção, etc.) que possuir.

No entanto, como mostrado na Figura 4.2, os conjuntos de teste QMap e QSet apresentam os maiores tempos durante o processo de verificação ao se utilizar o ESBMC++, apesar de QVector ser o mais extenso conjunto de casos de teste existente. Isso acontece por ser analisado a quantidade de laços presente no programa e não somente o número de linhas de código que ele possui, o que afeta diretamente os tempos de verificação. Na realidade, as estruturas internas associadas ao modelo operacional das classes *QMap* e *QSet* contêm mais laços do que as demais, desta forma, obtém-se tempos de verificação mais longos. Como também visto na Figura 4.2, o LLBMC apresenta um maior tempo de verificação ao se utilizar o conjunto de teste QVector, devido a 2 casos de teste em que houve estouro do tempo estimado para que seja realizada a verificação. Além disso, o DIVINE é a ferramenta que apresenta o menor desempenho entre as citadas, pois seu processo de criação do *bytecode* é mais custoso do que a realização da verificação sobre o mesmo. Dessa forma, os conjuntos de teste com mais programas a serem analisados obtiveram os maiores tempos que no caso são QVector, QList, QMap, QLinkedList

e QSet.

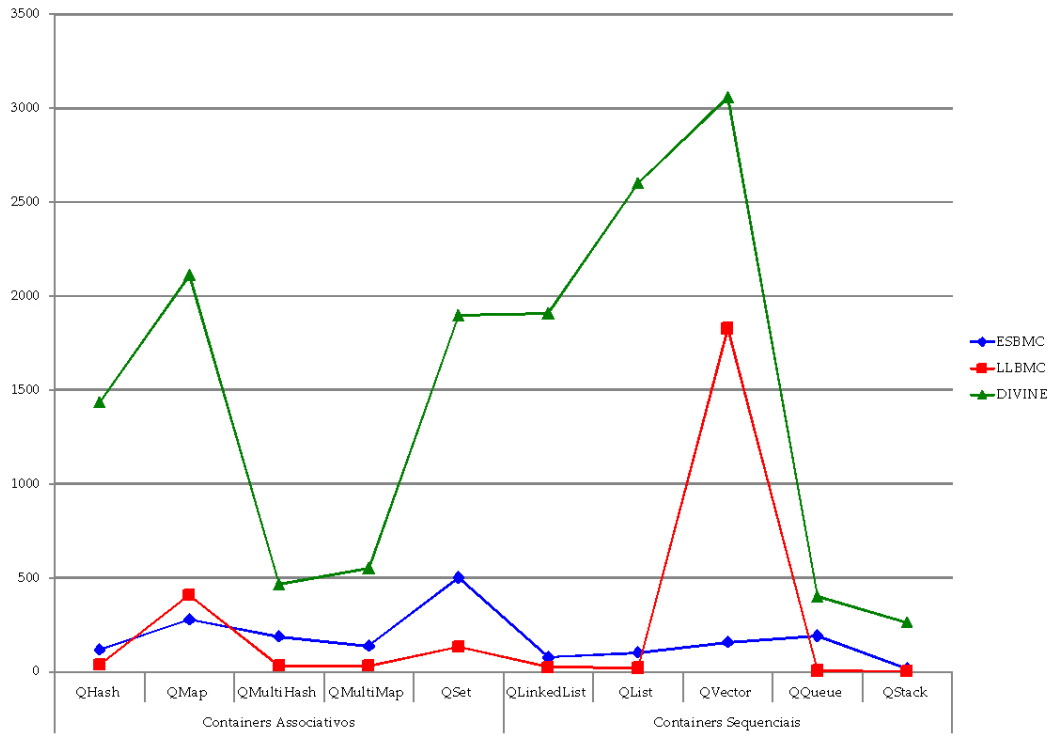


Figura 4.2: Comparação entre os tempos de verificação em relação a ESBMC++, LLBMC e DIVINE.

A Figura 4.3 mostra que todos os verificadores obtiveram uma taxa de corbetura acima de 80% para os *containers* do tipo associativo. Contudo, LLBMC não se manteve com a mesma taxa ao analisar os *containers* do tipo sequencial. Vale ressaltar que todos os casos de teste dos conjuntos de teste QMultiMap e QMultiHash foram verificados corretamente por todos os verificadores utilizados. Os conjuntos de teste QHash, QMap e QSet, por sua vez, apresentaram uma taxa média de $6,7\%$ para resultados falsos positivos (FP) e falsos negativos (FN), ou seja, de 3 a 18 casos de teste dos 267 casos de teste são considerados falsos positivos (FP) ou falsos negativos (FN) devido às limitações na representação interna dos *iterators*. Além disso, LLBMC e DIVINE, respectivamente, não conseguiram verificar cerca de $4,5\%$ e $13,9\%$ dos casos de teste dos *containers* associativos, ou seja, 12 e 13 dos 267 casos de teste não podem ser verificados devido as falhas no processo de criação do *bytecode*. Em relação aos *containers* do tipo sequencial, LLBMC apresentou baixas taxas de cobertura para os conjuntos de teste QVector, QLinkedList e QList, taxas a cerca de $67,7\%$ a 77%, ou seja, 84/117 dos 124/152 casos de teste, respectivamente. Além disso, cerca de $22,9\%$ dos casos de teste, 83 dos 363 casos de teste analisados apresentaram resultados falsos negativos (FN) devido também a

problemas com a representação interna dos *iterators*. Vale ressaltar, que cerca de 5% dos casos de teste, 18 dos 363 casos de teste analisados não haviam sido verificados por LLBMC, uma vez que não foi capaz de criar os *bytecodes* desejados. O ESBMC++ e DIVINE, por sua vez, apresentaram uma taxa de erro de no máximo de $6,6\%$, ou seja, 24 dos 363 casos de teste para os conjuntos de teste QVector, QLinkedList e QList apresentam erros de análise em suas pós-condições. Além disso, todos os casos de teste dos conjuntos de teste QQueue e QStack foram verificados corretamente, com exceção de dois casos presentes em QStack, pois ao se utilizar o Boolector com a ferramenta ESBMC++, não foi possível obter uma solução para as fórmulas SMT criadas para os casos analisados.

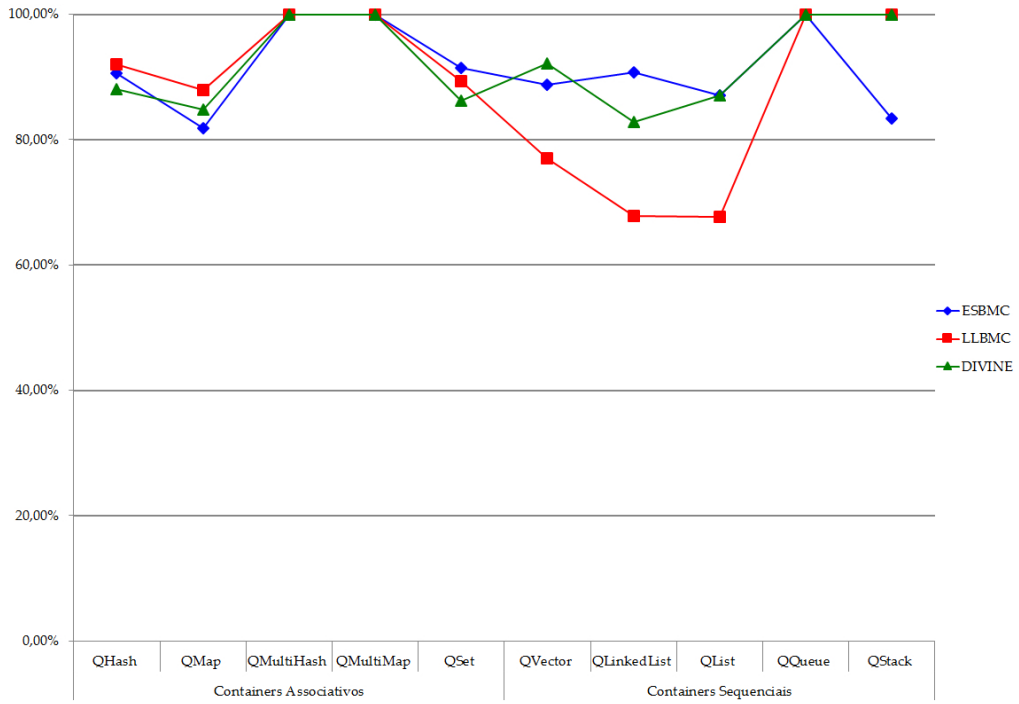


Figura 4.3: Comparação entre a taxa de cobertura em relação a ESBMC++, LLBMC e DIVINE.

Os conjuntos de teste QList, QMap, QVector e QSet possuem mais resultados falsos positivos (FP) e falsos negativos (FN) em seus testes ao se utilizar o ESBMC++ do que as outras ferramentas. A taxa de cobertura a cerca dos casos de teste verificados corretamente se encontra em torno de 80% a 90%, respectivamente, o que demonstra a eficácia em relação a verificação realizada uma vez que cada caso de teste verifica características diferentes dos *containers* sequenciais e associativos.

Vale ressaltar que o ESBMC++ foi capaz de identificar cerca de $89,3\%$ dos erros nos casos de teste utilizados (*i.e.*, 631 dos 708 casos de testes utilizados possuem erros), o que

demonstra também a eficácia do método proposto. Similarmente, LLBMC e DIVINE apresentam, respectivamente, taxas com 81,4% e 89% (*i.e.*, 576 e 630 dos 708 casos de teste utilizados possuem erros), o que também demonstra uma boa adequação do modelo operacional (QtOM) junto a outras ferramentas de verificação. Como consequência, o método proposto não apenas se limita a uma determinada ferramenta, podendo adaptar-se para aplicações específicas em que alguma abordagem seja a mais adequeda do que outra.

4.4 Verificação dos resultados para aplicações reais

Dado que o conjunto de casos de teste proposto possui como objetivo verificar propriedades específicas dos módulos pertencentes ao *framework* Qt, também é necessário incluir resultados de verificações que envolvam aplicações reais. Os parágrafos seguintes descrevem as respectivas aplicações e seus resultados associados.

A aplicação chamada *Locomaps* [20] é um exemplo de programa que utiliza o *framework* Qt e exibe imagens de satélite, terrenos, mapas de ruas, serviço de planejamento de mapas em mosaicos e possui um integração com GPS *Qt Geo*. Utilizando o mesmo código fonte esta aplicação pode ser compilada e executada nos principais sistemas operacionais existentes (Mac OS X, Linux e Windows). Está aplicação possui duas classes com 115 linhas de código no total, utilizando Qt/C++ e usando cinco APIs diferentes do framework Qt (*QApplication*, *QCoreApplication*, *QDesktopWidget*, *QtDeclarative* e *QMainWindow*). Vale mencionar que o código escrito em Qt/C++ desta aplicação, as APIs e as bibliotecas utilizadas são considerados no processo de verificação, assim como, as propriedades relacionadas a eles.

ArcGIS [50] para as forças armadas é uma plataforma geográfica que é utilizado para criar, organizar e compartilhar materiais geográficos com usuários que utilizam mapas inteligentes *online*. A partir disso, *GeoMessage Simulator* [21] possui como entrada de dados arquivos XML e cria em diferentes frequências datagramas utilizando o protocolo de datagramas por usuário (*em inglês*, User Datagram Protocol - UDP) para aplicações ArcGIS e componentes do sistema. *GeoMessage Simulator* também é uma aplicação multi-plataforma que contém 1209 linhas de códigos em Qt/C++ que utiliza 20 diferentes APIs do *framework* Qt englobando várias características, tais como o sistema de eventos de Qt, *strings*, manipulação de arquivos e *widgets*. Vale ressaltar que *GeoMessage Simulator* usa duas classes, *QMutex* e *QMutexLocker*, relacionadas ao módulo *Qt Threading* que possui classes para programas con-

correntes. Tais classes foram utilizadas na aplicação para travar ou destravar mutexes e, o mais importante, ESBMC++ é capaz de verificar adequadamente esses tipos de estruturas. No entanto, o modelo operacional (QtOM) ainda não fornece um suporte completo para o módulo *Qt Threading*.

O ESBMC++ junto com QtOM foi aplicado para verificar as aplicações *Locomaps* e *GeoMessage Simulator* buscando verificar as seguintes propriedades: violação dos limites de um array, estouros aritméticos, divisão por zero, segurança de ponteiro e outras propriedades específicas do *framework* definidas em QtOM de acordo com o Capítulo 3. Além disso, ESBMC++ foi capaz de identificar completamente o código-fonte de cada aplicação, utilizando cinco diferentes módulos de QtOM para *Locomaps* e vinte módulos para *GeoMessage Simulator*, ou seja, cada módulo de QtOM usado correspondia a uma API utilizada pela aplicação que seria verificada. O processo de verificação de ambas as aplicações foi totalmente automático e o método proposto levou aproximadamente $6,7$ segundos para gerar 32 VCs para *Locomaps* e 16 segundos para gerar 6421 VCs para *GeoMessage Simulator* em um *desktop* comum. Além disso, o ESBMC++ não relata caso haja qualquer falso negativo, mas foi capaz de encontrar bugs semelhantes em ambas as aplicações, as quais foram confirmadas pelos desenvolvedores e são explicadas abaixo.

```
1 int main(int argc, char *argv[]) {
2   QApplication app(argc, argv);
3   return app.exec();
4 }
```

Figura 4.4: Fragmento de código do arquivo principal da aplicação *Locomaps*.

A Figura 4.4 mostra um fragmento de código retirado do arquivo principal da aplicação *Locomaps* que utiliza a classe *QApplication* que está presente no módulo *QtWidgets*. Nesse caso em particular, se o parâmetro *argv* não for corretamente inicializado, logo o construtor ao ser chamado pelo objeto *app* não é executado de forma correta acarretando em falhas na aplicação (veja a linha 2, na Figura 4.4). A fim de verificar esta propriedade, o ESBMC++ analisa 2 assertivas em relação aos parâmetros de entrada da aplicação (veja as linhas 4 e 5, na Figura 4.5), avaliando-as como pré-condições. Um erro semelhante também foi encontrado na aplicação *GeoMessage Simulator* e uma maneira possível para corrigir tal erro é sempre verificar, com instruções condicionais, se *argv* e *argc* são argumentos válidos antes de utilizalas em uma operação.

```
1 class QApplication {
2   ...
3   QApplication( int & argc, char ** argv ){
4   __ESBMC_assert(argc > 0, ``Invalid parameter'');
5   __ESBMC_assert(argv != NULL, ``Invalid pointer'');
6   this->str = argv;
7   this->_size = strlen(*argv);
8   ...
9  }
10  ...
11 };
```

Figura 4.5: Modelo operacional para o construstor de *QApplication*().

4.5 Verificação da conformidade de QtOM

Todos os processos, funções, APIs, ferramentas e especificações necessárias para o desenvolvimento de aplicações que utilizam o *framework* Qt são descritas por *The Qt Developers* em [51]. Na verificação de modelos, as aplicações analisadas se utilizam de funções disponibilizadas pelo ambiente de verificação criado com o objetivo de garantir a exatidão da verificação, ou seja, se obter uma análise de forma correta. Para que isso ocorra, o comportamento do sistema desenvolvido (modelo operacional) deve ser o mesmo do sistema real (*framework*) analisado.

Sendo assim, inicialmente foi realizado uma análise sobre todos os casos de testes presentes em *esbmc–qt* com o objetivo de verificar a imparcialidade e a confiabilidade das aplicações utilizadas. Todos os casos de teste foram desenvolvidos utilizando a linguaguem C++ junto com as funções disponibilizadas pelo *framework* Qt, onde cada implementação usa as estruturas de dados e funções presente em Qt *containers*, em todas as possíveis condições e de acordo com *The Qt Documentation* [48]. Cada caso de teste proporciona uma falha e um sucesso de acordo com a estrutura ou função empregada como mostrado na Figura 4.6. Além do mais, todos os casos de testes presentes em *esbmc–qt* foram compilados e executados utilizando o ambiente de desenvolvimento do *framework* Qt denominado QtCreator com o objetivo de se obter um comportamento base do *framework* em relação a Qt *containers*. QtCreator é um ambiente de desenvolvimento integrado (IDE) multiplataforma utilizado por desenvolvedores para criar diversas aplicações que utilizam o *framework* Qt para desktops, sistemas embarcados e plataformas de dispositivos móveis. É um ambiente de desenvolvimento disponível para Linux, OS X e Windows [52]. O mesmo foi realizado com o compilador Clang junto com o modelo

operacional (QtOM), cujo objetivo era de determinar o comportamento de QtOM. Clang é um compilador *front end* para as linguaguens C e C++ e foi utilizado para criar o nível de conformidade referente a QtOM, pois já havia sido utilizado em testes anteriores (ver Seção 4.3). Por fim, foi realizado a comparação entre o comportamento base do *framework* Qt em relação ao comportamento do modelo operacional (QtOM) com o objetivo de analisar o nível de conformidade entre ambos. Essa comparação é mostrada seguindo o diagrama de atividade presente na Figura 4.7.

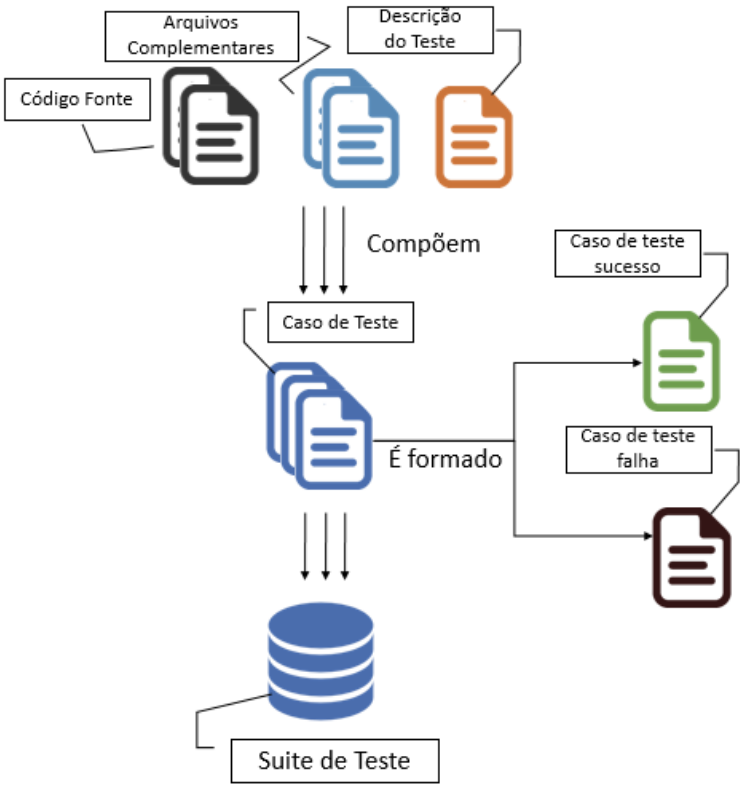


Figura 4.6: Como os casos de teste presente em *esbmc–qt* estão compostos e formados.

A Figura 4.8 mostra o resultado da comparação descrita anteriormente, assim como, em testes anteriores os conjuntos de testes estão divididos em *containers* associativos compostos por QHash, QSet, QMap, QMultiHash e QMultiMap e *containers* sequenciais compostos por QList, QQueue, QLinkedList, QStack e QVector. Vale ressaltar que por QtCreator ser o ambiente de desenvolvimento padrão do *framework* Qt, obteve um nível de conformidade de 100% em todos os conjuntos de testes que compõe *esbmc–qt*, em todas as condições possíveis de acordo com *The Qt Documentation* [48]. Entretanto, ao se analisar o comportamento do compilador Clang junto com o modelo operacional proposto (QtOM), se obtem um nível de conformidade com média de 97,827% em todos os conjuntos de testes que compõe *esbmc–qt*

em todas as condições possíveis de acordo com *The Qt Documentation* [48].

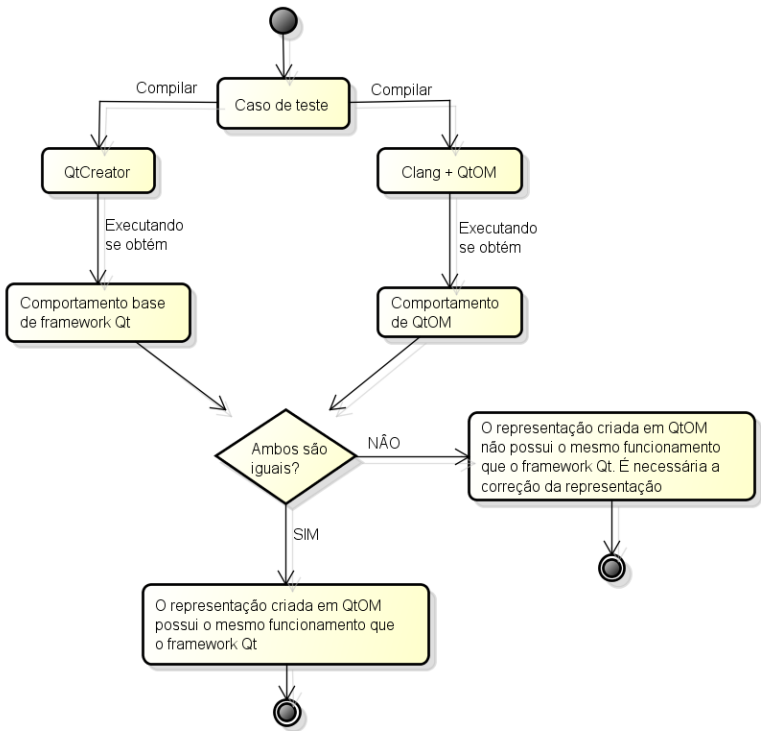


Figura 4.7: Diagrama de atividade referente a verificação da conformidade de QtOM.

Ao se analisar de modo particular, os conjuntos de testes QMultiHash, QMultiMap, QQueue, QStack obtiveram um nível de 100% de semalhança comparado aos resultados obtidos por QtCreator. O conjunto de teste QMap obteve o menor nível de semelhança de 91,919% em relação a QtCreator, isso ocorreu devido algumas representações presente no modelo operacional possuirem erros no uso de seus iterators o que impossibilitava o acesso de determinados dados nas estruturas representadas e nas representações que utilizam funções estáticas, na qual, havia a perda do dado utilizado na estrutura representada, QMap é considerado o conjunto de teste com o maior número de falsos positivos e negativos entres os conjuntos de teste existentes correspondendo a cerca de 8,080% dos seus casos de teste. Os conjuntos de testes QHash, QLinkedList e QList, respectivamente, obtiveram níveis de semelhança de 94,667%, 98,850%, 97,580%, isso ocorreu devido a problemas com a representação de iterators usadas no modelo operacional, assim como, em QMap ao se utilizar mais de um iterator no caso de teste, a referência sobre os demais iterators utilizados é perdida no momento em que se usa um iterator para acessar/modificar dados presente no *container* analisado. O conjunto de teste QLinkedList é o conjunto de teste que obteve o maior nível de semelhança entre os conjuntos de teste que não obtiveram um nível de conformidade de 100%. O conjunto de teste QSet obteve um nível de

semelhança de 97,872% em relação a QtCreator, isso ocorreu devido algumas representações possuirem problemas em suas funções estáticas havendo perda de dados nas estruturas representadas. Por fim, o conjunto de caso QVector considerado o maior conjunto em número de casos de teste obteve um nível de semalhança de 97,378%, pois não foi possível realizar a representação de funções onde em suas definições tinham como objetivo realizar algum tipo de manipulação da memória como "Ajuste fino" "Melhoria no gerenciamento da memória" (e.g. a função tem como aplicabilidade armazenar ou liberar uma quantidade da memória utilizada pelo o programa para o melhor uso do *container*) entre outras definições. Vale ressaltar que não é possível criar esses tipos de representação, pois não se sabe como o *framework* analisado trata essas funções internamente o que impossibilita criar uma representação na linguagem base utilizada. O conjunto QVector também teve problemas relacionados a utilização de mais um iterator, assim como, os conjuntos QMap, QHash, QLinkedList e QList. Como proposta para os erros encontrados a nível de conformidade, o uso de uma estrutura que fizesse o armazenamento dos dados mesmo que de forma temporária poderia auxiliar as aplicações tanto estáticas como as que se utilizam mais de um iterator no manuseio interno dos dados vindos da aplicação, assim evitando a perda de referência em ambos os problemas encontrados.

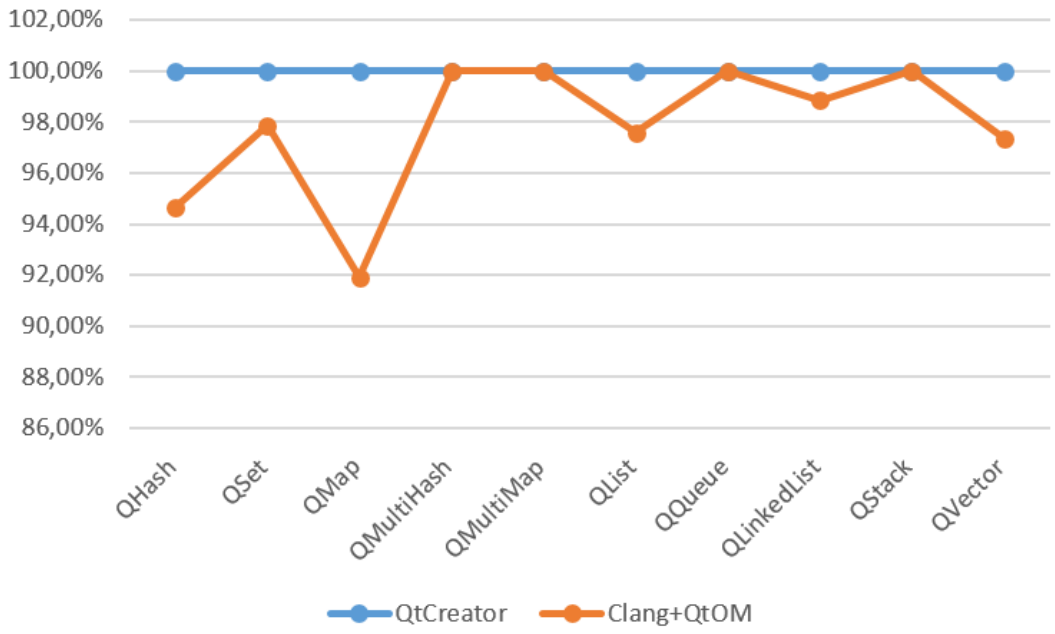


Figura 4.8: Comparação entre o comportamento de QtCreator e Clang+QtOM parar medir a conformidade de QtOM.

4.6 Resumo

De acordo com o embasamento teórico realizado pelos capítulos anteriores em relação as técnicas SMT utilizadas, o processo de verificação de ESBMC++ junto ao modelo operacional (QtOM). Neste capítulo, inicialmente foi descrito como foi construída a configuração dos experimentos realizados, com o intuito de avaliar a eficácia do método proposto acerca da verificação de programas que utilizam o *framework* Qt, a partir de uma suíte de teste denomidada *esbmc–qt* que contém todos os casos de teste utilizadas na atual avaliação. A suíte *esbmc–qt* contém 711 programas Qt/C++ que de acordo com os experimentos realizados é possível garantir que 353 dos 711 casos de teste contêm erro (*i.e.*, 49.65% do casos de teste analisados contêm erro) e 358 casos de teste não possuem falhas (*i.e.*, 50.35% dos casos de teste analisados são considerados corretos).

Em seguida, foi descrito como foi realizado e avaliado a comparação entre os solucionadores SMT utilizados (Z3, Boolector e Yices 2) junto com o ESBMC++ utilizando programas Qt/C++ sequenciais. Isso foi realizado, pois solucionadores SMT podem afetar fortemente os resultados obtidos, uma vez que não existe homogeneidade em relação a abordagem de implementação e respectivas heurísticas. Sendo assim, Yices 2 obteve os piores resultados, apresentando uma taxa de cobertura de 78% e um tempo de verificação de $26,27$ minutos. Entretanto, Z3 e Boolector apresentaram uma taxa de cobertura de 89% com tempos de verificação de $223,6$ minutos e $26,38$ minutos, respectivamente. Por fim, Boolector mostrou-se $8,5$ vezes mais rápido do que o solucionador Z3, apesar da mesma precisão de ambos.

Também foram descritos os resultados obtidos a cerca da verificação de todos os casos de teste presentes em *esbmc–qt* utilizando ferramentas de verificação distintas (ESBMC++, LLBMC e DIVINE) com o objetivo de analisar a versatilidade e a eficácia de QtOM junto a outras abordagens de verificação. Cada processo de verificação e as limitações das ferramentas são descritos. Vale ressaltar que ESBMC++ foi capaz de identificar cerca de $89,3$% dos erros nos casos de teste utilizados (*i.e.*, 631 dos 708 casos de testes utilizados possuem erros), LLBMC e DIVINE apresentam, respectivamente, taxas com $81,4$% e 89% (*i.e.*, 576 e 630 dos 708 casos de teste utilizados possuem erros). Como consequência, o método proposto não apenas se limita a uma determinada ferramenta, podendo-se adaptar para aplicações específicas em que algumas abordagens sejam mais adequedas do que outras.

Também foi descrito os resultados obtidos atráves da verificação de duas aplicações

reais *Locomaps* e *GeoMessage Simulator* utilizando QtOM. ESBMC++ junto ao modelo operacional (QtOM) foi aplicado para verificar as aplicações *Locomaps* e *GeoMessage Simulator* buscando verificar as seguintes propriedades: violação dos limites de um array, estouros aritméticos, divisão por zero, segurança de ponteiro e outras propriedades específicas do *framework* definidas em QtOM. O processo de verificação de ambas as aplicações com a metodologia proposta levou aproximadamente $6,7$ segundos para gerar 32 VCs para *Locomaps* e 16 segundos para gerar 6421 VCs para *GeoMessage Simulator* em um comum *desktop* onde ambas aplicações possuiam o mesmo erro.

Por fim, é descrito os resultados obtidos atráves da comparação entre o comportamento real do *framework* Qt e o comportamento do modelo operacional (QtOM) com o objetivo de avaliar o nível de conformidade entre ambos. Primeiramente, houve uma análise sobre todos os casos de testes presentes em *esbmc–qt* com o intuito de verificar a imparcialidade e confiabilidade das aplicações. Para se obter o comportamento real do *framework* Qt se utilizou um ambiente de desenvolvimento padrão denominado QtCreator. Todos os casos de testes presente em *esbmc–qt* foram compilados e executados com QtCreator que por ser um ambiente de desenvolvimento padrão do *framework* analisado obteve um nível de conformidade de 100% sobre todos os casos de testes submetido. Logo em seguida, o mesmo foi realizado utilizando o compilador Clang junto com QtOM, dessa forma, se obteve o comportamento do modelo operacional (QtOM) com uma média de 97,827% de nível de conformidade. Vale ressaltar que todo caso de teste presente em *esbmc–qt* abordam todas as condições possíveis de acordo com *The Qt Documentation*. De modo particular, os conjuntos de teste QHash, QSet, QMap, QMultiHash, QMultiMap, QList, QQueue, QLinkedList, QStack, QVector obtiveram níveis de conformidade de 94,667%, 97,872%, 91,919%, 100,000%, 100,000%, 97,580%, 100,000%, 98,850%, 100,000%, 97,378%, respectivamente. Cada conjunto de teste possui suas peculiaridades, sendo QMap o conjunto de teste que obteve o menor nível de conformidade devido a problemas em suas representações e QLinkedList sendo o conjunto que obteve o maior nível de conformidade entre os conjuntos que não obtiveram um nível igual a 100%.

Capítulo 5

Conclusões

A abordagem proposta neste trabalho tem como objetivo verificar programas que utilizam o *framework* multiplataforma Qt desenvolvido em Qt/C++, usando um modelo operacional denominado de QtOM que se utiliza de pré e pós-condições, simulação de características (por exemplo, como os elementos que possuem valores são manipulados e armazenados) e também da forma como são utilizados em dispositivos de eletrônica de consumo. A forma como o modelo operacional proposto foi implementado, foi também descrita, levando-se em consideração que é usado para verificar *containers* de tipos sequenciais e associativos. Além disso, uma aplicação *touchscreen* baseada em Qt que utiliza mapas de navegação, imagens de satélite e dados de terrenos [20] e outra que gera *datagramas broadcast* UDP com base em arquivos XML [21] foram verificadas usando o ESBMC++ com o QtOM. Desta forma, foi mostrado o potencial da abordagem proposta para a verificação de aplicações reais baseadas no *framework* Qt.

Este trabalho possui como contribuição principal a construção de um modelo operacional denominado QtOM que oferece suporte à *containers* sequenciais e associativos que utilizam o *framework* multiplataforma Qt. Os experimentos que foram realizados envolvem programas em Qt/C++ com características oferecidas pelas classes *container* do *framework* multiplataforma Qt e abordam todas as condições possíveis de acordo com *The Qt Documentation* [48].

Além disso, também foram avaliados o desempenho dos solucionadores Z3, Boolector e Yices 2, dado que eles foram utilizados no processo de verificação dos programas que utilizam o *framework* multiplataforma Qt com ESBMC++ e o modelo operacional proposto

(QtOM). Como resultado, o Boolector apresentou a maior taxa de cobertura dos programas verificados com um menor tempo de verificação e o ESBMC++ apresentou uma taxa de sucesso de 89,3% para o conjunto de teste utilizado com um tempo de 1760 segundos de verificação. Ainda foi avaliado o nível de conformidade do modelo operacional proposto (QtOM) atráves da comparação de entre o comportamento real do *framework* e o comportamento do modelo operacional desenvolvido. Como resultado, QtOM obteve um nível de conformidade com média de 97,827% e de maneira particular os conjuntos de teste QHash, QSet, QMap, QMultiHash, QMultiMap, QList, QQueue, QLinkedList, QStack, QVector obtiveram níveis de conformidade de 94,667%, 97,872%, 91,919%, 100,000%, 100,000%, 97,580%, 100,000%, 98,850%, 100,000%, 97,378%, respectivamente. O conjunto de teste QMap obteve o menor nível de conformidade devido a problemas em suas representações e QLinkedList foi o conjunto de teste que obteve o maior nível de conformidade entre os conjuntos que não obtiveram um nível igual a 100%.

Por fim, outra fundamental contribuição é a integração de QtOM dentro do processo de verificação de outro dois diferentes verificadores de modelo conhecidos como LLBMC e DIVINE, demonstrando a versatilidade do QtOM. Esse tipo de alternativa também demonstra resultados importantes, uma vez que LLBMC detectou 95% dos erros existentes com um tempo de verificação de 2513 segundos e o DIVINE encontrou 92% dos erros existentes com um tempo de verificação em torno de 1760 segundos. Contudo, o LLBMC possui a maior taxa de resultados incorretos entre as ferramentas utilizadas com 18,6%, seguido de DIVINE com 11,3% e por fim ESBMC++ com 10,9%. Vale ressaltar que o DIVINE é 7 vezes mais lento do que as demais ferramentas utilizadas, seguido pelo LLBMC e ESBMC++, respectivamente. Em resumo, o QtOM pode ser integrado em uma ferramenta de verificação adequada e utilizado para verificar programas reais que estão escritos em C++/Qt para cenários específicos e aplicações.

5.1 Trabalhos Futuros

Como trabalhos futuros, o modelo operacional proposto (QtOM) será estendido com o objetivo de oferecer suporte à verificação de programas multi-tarefas que utilizam o *framework* multiplataforma Qt. Além disso, mais classes e bibliotecas serão adicionadas com o intuito de aumentar a cobertura da verificação em relação ao *framework* multiplataforma Qt e dessa forma validar suas respectivas propriedades. Novas ferramentas para medição de desempenho do

modelo operacional proposto e novos testes de conformidade serão incluídos, a fim de verificar se as rotinas específicas estão de acordo com as limitações de tempo. Por fim, será investigada a verificação por indução matemática no ESBMC^{QtOM} para provar a corretude de programas Qt [53, 54].

Referências Bibliográficas

[1] CORDEIRO, L.; FISCHER, B.; CHEN, H.; MARQUES-SILVA, J. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In: *Proceedings of the 2009 International Conference on Embedded Software and Systems*. 2009. (ICESS '09), p. 396–403. ISBN 978-0-7695-3678-1.

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

[3] BERARD, B.; BIDOIT, M.; FINKEL, A.; LAROUSSINIE, F.; PETIT, A.; PETRUCCI, L.; SCHNOEBELEN, P. *Systems and Software Verification: Model-Checking Techniques and Tools*. : Springer Publishing Company, Incorporated, 2010. ISBN 3642074782, 9783642074783.

[4] CLARKE JR., E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking*. : MIT Press, 1999. ISBN 0-262-03270-8.

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

[6] MEHLITZ, P.; RUNGTA, N.; VISSER, W. A hands-on java pathfinder tutorial. In: *Proceedings of the 2013 International Conference on Software Engineering*. 2013. p. 1493–1495. ISBN 978-1-4673-3076-3.

[7] MERWE, H. van der; MERWE, B. van der; VISSER, W. Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes*, 2014.

[8] MERWE, H. van der; TKACHUK, O.; MERWE, B. van der; VISSER, W. Generation of library models for verification of android applications. *ACM SIGSOFT Software Engineering Notes*, 2015.

[9] RAMALHO, M.; FREITAS, M.; SOUSA, F.; MARQUES, H.; CORDEIRO, L.; FISCHER, B. Smt-based bounded model checking of c++ programs. In: *Proceedings of the 20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems*. 2013. p. 147–156.

[10] MONTEIRO, F. R.; CORDEIRO, L. C.; FILHO, E. B. de L. Bounded Model Checking of C++ Programs Based on the Qt Framework. In: *4th Global Conference on Consumer Electronics*. : IEEE, 2015.

[11] 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.

[12] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 2012.

[13] PEREIRA, P. A.; ALBUQUERQUE, H. F.; MARQUES, H. M.; SILVA, I. S.; CARVALHO, C. B.; CORDEIRO, L. C. Verifying cuda programs using smt-based context-bounded model checking. ACM, New York, NY, USA, p. 1648–1653, 2016. Disponível em: <http://doi.acm.org/10.1145/2851613.2851830>.

[14] PEREIRA, P.; ALBUQUERQUE, H.; SILVA, I. da; MARQUES, H.; MONTEIRO, F.; FERREIRA, R.; CORDEIRO, L. Smt-based context-bounded model checking for cuda programs. *Concurrency and Computation: Practice and Experience*, p. n/a–n/a, 2016. ISSN 1532-0634. Disponível em: <http://dx.doi.org/10.1002/cpe.3934>.

[15] MOURA, L. M. de; BJØRNER, N. Z3: An efficient SMT solver. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2008. (TACAS'08/ETAPS'08), p. 337–340. ISBN 3-540-78799-2, 978-3-540-78799-0. Disponível em: <http://dl.acm.org/citation.cfm?id=1792734.1792766>.

[16] 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*. Cham: Springer International Publishing, 2014. p.

737–744. ISBN 978-3-319-08867-9. Disponível em: <http://dx.doi.org/10.1007/978-3-319-08867-9$_4$9>.

[17] BRUMMAYER, R.; BIERE, A. Boolector: An efficient SMT solver for bit-vectors and arrays. In: ____. *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 174–177. ISBN 978-3-642-00768-2. Disponível em: $<http://dx.doi.org/10.1007/978-3-642-00768-2_16>$.

[18] 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: *Computer Aided Verification (CAV 2013)*. 2013. p. 863–868.

[19] FALKE, S.; MERZ, F.; SINZ, C. The bounded model checker LLBMC. In: *ASE*. 2013. p. 706–709.

[20] Locomaps. *Spatial Minds and CyberData Corporation*. 2012. `https://github.com/craig-miller/locomaps`. [Online; accessed 10-September-2015].

[21] Environmental Systems Research Institute. *GeoMessage Simulator*. 2015. `https://github.com/Esri/geomessage-simulator-qt`. [Online; accessed 15-September-2015].

[22] KROENING, D.; TAUTSCHNIG, M. CBMC - C bounded model checker - (competition contribution). In: *TACAS*. 2014. p. 389–391.

[23] WANG, W.; BARRETT, C.; WIES, T. Cascade 2.0. In: *VMCAI*. 2014. p. 142–160.

[24] LATTNER, C. *CLang Documentation*. 2015. [Online; accessed December-2015].

[25] BLANC, N.; GROCE, A.; KROENING, D. Verifying C++ with STL containers via predicate abstraction. In: *ASE*. 2007. p. 521–524.

[26] Wintersteiger, C. *goto-cc – a C/C++ front-end for Verification*. 2009. `http://www.cprover.org/goto-cc/`. [Online; accessed January-2016].

[27] SITES, R. L. *Some Thoughts on Proving Clean Termination of Programs.* 1974.

[28] MERZ, F.; FALKE, S.; SINZ, C. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: *VSTTE*. 2012. p. 146–161.

[29] MORSE, J.; CORDEIRO, L.; NICOLE, D.; FISCHER, B. Model checking ltl properties over ansi-c programs with bounded traces. 2013.

[30] MORSE, J.; RAMALHO, M.; CORDEIRO, L.; NICOLE, D.; FISCHER, B. Esbmc 1.22. In: *Tools and Algorithms for the Construction and Analysis of Systems*. : Springer Berlin Heidelberg, 2014. p. 405–407.

[31] MORSE, J.; CORDEIRO, L.; NICOLE, D.; FISCHER, B. Handling unbounded loops with esbmc 1.20. In: *Tools and Algorithms for the Construction and Analysis of Systems*. : Springer Berlin Heidelberg, 2013. p. 619–622.

[32] CORDEIRO, L.; MORSE, J.; NICOLE, D.; FISCHER, B. Context-bounded model checking with esbmc 1.17. In: *Tools and Algorithms for the Construction and Analysis of Systems*. : Springer Berlin Heidelberg, 2012. p. 534–537.

[33] 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: *IFM*. 2012. p. 128–142.

[34] BRADLEY, A. R.; MANNA, Z. *The Calculus of Computation: Decision Procedures with Applications to Verification*. : Springer-Verlag New York, Inc., 2007. ISBN 3540741127.

[35] MCCARTHY, J. Towards a mathematical science of computation. In: *In IFIP Congress*. 1962. p. 21–28.

[36] BARRETT, C.; TINELLI, C. CVC3. In: *CAV*. 2007. p. 298–302.

[37] MOURA, L. M. de; BJØRNER, N. Satisfiability modulo theories: An appetizer. In: *SBMF*. 2009. p. 23–36.

[38] Qt Jambi. *Qt Jambi*. 2015. `http://qtjambi.org`. [Online; accessed December-2015].

[39] Qt in Home Media. 2011. `http://qt.nokia.com/qt-in-use/qt-in-home-media`. [Online; accessed July-2011].

[40] Qt in IP Communications. 2011. http://qt.nokia.com/qt-in-use/qt-in-ip-communications. [Online; accessed July-2011].

[41] Panasonic selects Qt for HD video system. 2011. http://qt.nokia.com/about/news/panasonic-selects-qt-for-hd-video-system. [Online; accessed July-2011].

[42] RESEARCH2GUIDANCE. *Cross-Platform Tool Benchmarking*. 2014.

[43] The Qt Company Ltd. *Signals and Slots - QtCore 5*. 2015. https://doc.qt.io/qt-5/signalsandslots.html. [Online; accessed 2-April-2015].

[44] The Qt Company Ltd. *The Meta-Object System*. 2015. http://doc.qt.io/qt-5/metaobjects.html. [Online; accessed 2-April-2015].

[45] MUSUVATHI, M.; PARK, D. Y. W.; CHOU, A.; ENGLER, D. R.; DILL, D. L. Cmc: A pragmatic approach to model checking real code. ACM, 2002.

[46] ISO/IEC. *ISO/IEC 14882:2003: Programming languages: C++*. : International Organization for Standardization, 2003.

[47] DEITEL, P.; DEITEL, H. *C++ How to Program*. : Prentice Hall, 2013. ISBN 978-85-7605-056-8.

[48] Qt Documentation. *Qt Documentation*. 2015. http://doc.qt.io/. [Online; accessed December-2015].

[49] The Open Group. *The Single UNIX ®Specification, Version 2 – time.h*. 1997. http://pubs.opengroup.org/onlinepubs/007908775/xsh/time.h.html. [Online; accessed December-2015].

[50] Environmental Systems Research Institute, Inc. *ArcGIS for the Military*. 2015. http://solutions.arcgis.com/military/. [Online; accessed 25-November-2015].

[51] Qt Developers. *Qt Developers*. 2015. https://www.qt.io/developers/. [Online; accessed July-2016].

[52] Qt Creator. *Qt Creator*. 2015. http://doc.qt.io/qtcreator/index.html. [Online; accessed July-2016].

[53] GADELHA, M. Y. R.; ISMAIL, H. I.; CORDEIRO, L. C. Handling loops in bounded model checking of c programs via k-induction. *International Journal on Software Tools for Technology Transfer*, p. 1–18, 2015. ISSN 1433-2787.

[54] ROCHA, H.; ISMAIL, H.; CORDEIRO, L. C.; BARRETO, R. S. Model checking embedded C software using k-induction and invariants. *V Brazilian Symposium on Computing Systems Engineering (SBESC)*, p. 90–95, 2015.

Apêndice A

Publicações

- **Garcia, M. A. P.**, Sousa, F. R. M., Cordeiro, L. C., Lima Filho, E. B. ESBMCQtOM: A Bounded Model Checking Tool to Verify Qt Applications. Software Testing, Verification and Reliability (submetido), John Wiley Sons Ltd, 2016.
- **Garcia, M. A. P.**, Sousa, F. R. M., Cordeiro, L. C., Lima Filho, E. B. ESBMCQtOM: A Bounded Model Checking Tool to Verify Qt Applications. In 23rd International SPIN symposium on Model Checking of Software (SPIN), LNCS 9641, pp. 97-103, 2016.