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

Fevereiro 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

Mário Angel Praia Garcia

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

Banca Examinadora

Prof. Ph.D. Lucas Carvalho Cordeiro – Presidente e Orientador
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
Abril de 2016

Resumo

O processo de desenvolvimento de software para sistemas embarcados tem crescido rapidamente, o que na maioria das vezes acarreta num 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 criar sistemas robustos e assim reduzir as taxas de *recall* de produtos. Além disso, a redução no tempo de desenvolvimento e na robustez dos sistemas criados podem ser alcançados através de *frameworks* multi-plataformas, tais como Qt, que oferece um conjunto de bibliotecas (gráficas) confiáveis para vários dispositivos. Desta forma, o trabalho atual propõe uma versão simplificada do *framework* Qt, integrado a um verificador baseado nas teorias do módulo da satisfatibilidade, denominado *Efficient SMT-Based Bounded Model Checker* (ESBMC++), o qual 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 proposta, também foi feito uma avaliação ultilizando outros verificadores que se encontram no estado da arte para programas em C++. Dessa maneira, evidenciando-se que a metodologia proposta é 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: Engenharia de Software, *Framework* Qt, Verificação Formal, *Bounded Model Checking*.

Abstract

The software development process 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. 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: software engineering, Qt Framework, bounded model checking, formal verification.

Capítulo 1

Introdução

A atual disseminação com relação aos sistemas embarcados e sua devida importância está de acordo com a evolução dos componentes de hardware e softwares associados a eles. Na realidade, esses sistemas têm crescido em relação a sua robustez e complexidade, onde se torna visível o uso de processadores com vários núcleos, memórias compartilhadas escaláveis entre outros avançados recursos, de maneira a suprir o crescimento do poder computacional exigido, no qual pode-se obter por meio da combinação de linguagens de programação e *frameworks*. Neste contexto, o Qt apresenta-se como um poderoso *framework* multi-plataforma para dispostivos, o qual enfatiza a criação de interface para usuário (IU) e o desenvolvimento de aplicações gráficas [1]. No entanto, como a complexidade de tais sistemas tende a crescer, o seu (bom) funcionamento se torna dependente do usuário; dependência está que também cresce de forma rápida. Em consequência, a confiabiliadade destes 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 mais investem 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 [2]. Entre as alternativas já existentes, uma das mais eficiente e menos custosa é a abordagem da verificação de modelos [3] [4]. 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 byte-code [5], mas não há suporte a verificação (completa) de aplicações Java que utilizam o sistema operacional Android [6]. Na realidade, esta verificação

somente se torna possível se existir uma representação abstrata das bibliotecas associadas, denominada de modelo operacional(MO), que de forma conservadora aproxima-se a semântica usada pelo sistema a ser verificado [7].

Este trabalho 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. 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++) [8] [9] [10], a fim de verificar as específicas propriedades de programas em Qt/C++. A combinação entre ESBMC e MOs foi aplicada anteriormente para que se houvesse suporte a programas em C++, conforme descrito por Ramalho *et al*. [8]. No entanto, na metodologia proposta, um MO é utilizado para identificar elementos do *framework* Qt e verificar propriedades específicas relacionadas a essas estruturas por meio de pré e pós-condições [11].

1.1 Descrição do Problema

Descrição

1.2 Objetivos

O objetivo geral deste trabalhar é aplicar as técnicas de verificação utilizando modelos operacionais para checar aplicações que utilizam o *framework* Qt, usando ferramentas de verificação, a fim de validar a cobertura e o comportamento do modelo operacional proposto em relação ao *framework* utilizado.

Os objetivos específicos são listados a seguir:

- Desenvolver uma implementação simplificada que oferece estritamente o mesmo comportamento do *framework* Qt com foco na verificação de suas propriedades.
- Verificar o acesso de memória inválida, valores que especificam o tempo, acesso à arquivos ausentes, ponteiros nulos, manipulação de cadeia de caracteres (*strings*), uso de

containers, entre outras propriedades de interesse para aplicações Qt.

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

1.3 Trabalhos relacionados

De acordo com a atual literatura sobre verificação, não existe outro verificador disponível que seja capaz de verificar 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*. Além disso, a verificação por meio de testes manuais é um processo árduo e custoso [11]. Em resumo, a técnica BMC aplicada em verificação de software é usada em diversos verificadores [12, 13, 10, 14] 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].

Ramalho *et al.* [8] 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 à verificação das características mais complexas que a linguagem oferece, como *containers* e tratamento de exceção. Vale ressaltar que os modelos 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.

Merz, Falke e Sinz [12] 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 nas bibliotecas de template padrão (do inglês, 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.* apresenta o DIVINE [16] como um verificador de modelo de estado explícito para programas ANSI-C/C++ sequencias 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 conhecido como Clang [17] como front-end, a fim de converter programas C++ em uma representação intermediária utilizada pelo LLVM, que logo em seguida realiza a verificação sobre o *byte-code* produzido. Embora o DIVINE possua uma implementação de ANSI-C e das bibliotecas padrões do C++, o que lhe permite verificar programas desenvolvidos com essas linguaguens, o mesmo não possui qualquer suporte à representação disponibilizada pelo *framework* Qt. De acordo com a abordagem proposta, DIVINE é capaz de criar um programa que possa ser interpretado totalmente pelo LLVM e então ser verificado logo em seguida.

Blanc, Groce e Kroening descrevem a verificação de programas em C++ que usam *containers* STL através de abstração de predicados [18], 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 3.1).

O *C Bounded Model Checker* (CBMC) implementa a técnica BMC para programas ANSI-C/C++ por meio de solucionadores SAT/SMT [13]. Vale ressaltar que ESBMC foi construído a partir do CBMC, portanto, ambos verificadores possuem processos de verificação semelhantes. Na verdade, CBMC processa programas C/C++ usando a ferramenta goto-cc [19], a qual compila o código fonte da aplicação para um *GOTO-programs* equivalente (ou seja, um

grafo de fluxo de controle), a partir de um modelo compátivel ao GCC. A partir de *GOTO-programs*, 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 *C* e *P* que registram as *restrições* (ou seja, premissas e atribuições de variáveis) e as *propriedades* (ou seja, condições de segurança 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 [20]. 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.* [8] e Merz *et al.* [21] relatam que o CBMC falha ao verificar diversas aplicações simples em C++, o que também foi confirmado neste trabalho (ver Seção 5.3).

Finalmente, destaca-se que o QtOM está completamente escrito na linguagem de programação C++, o que facilita a integração dentro de processos de verificação de outros verificadores. No presente trabalho, QtOM não foi somente intregado ao ESBMC++, mas também no DIVINE [16] e LLBMC [12], afim de obter uma avaliação mais justa sobre a abordagem proposta.

1.4 Contribuições

O trabalho é uma extensão de trabalhos já publicados anteriormente [11]. O respectivo modelo operacional(MO) proposto 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:

- Suportar *containers* baseados em templates sequenciais e associativos;
- Integrar o modelo operacional proposto denominado QtOM ao processo de verificação de programas em C++ em verificadores que se encontram no estado da arte, neste caso, DIVINE [16] e LLBMC [12];
- Fornecer suporte à verificação de duas aplicações baseadas em Qt denominadas Locomaps [22] e Geomessage [23], respectivamente.

- Avaliar o desempenho de três solucionadores SMT(Z3 [15], Yices [24] e Boolector [25]) sobre o conjunto de *benchmarks* utilizados extensivamente e ampliado em relação ao trabalho anterior [11] juntamente com a abordagem proposta.

Por fim, em particular, foram incluídas representações para todas as bibliotecas relacionadas com as classes de Qt *containers*. De acordo com o conhecimento atual em verificação de software, não há outro verificador que utilize modelos e se aplique técnicas BMC para verificar programas baseados no *framework* Qt sobre dispositivos de eletrônica de consumo.

1.5 Organização da Dissertação

Neste capítulo, inicialmente descreveram-se sobre o contexto que envolve o trabalho, a motivação, seus objetivos e 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 de 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.

- O Capítulo 4 descreve a implementação formal de Qt *Containers* associativos e sequencias desenvolvidos de forma detalhada.

- O Capítulo 5, descreve 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 [22] e a segunda aplicação cria um *broadcast User Datagram Protocol* (UDP) baseado em arquivos XML.

- Por fim, o Capítulo 6 apresenta as conclusões, destacando a importância da criação de um modelo para verificar aplicações que utilizam framework Qt, assim como, os trabalhos futuros também são descritos.

Capítulo 2

Fundamentação Teórica

Este capítulo descreve a arquitetura de ESBMC++ e algumas características estruturais do *framework* multiplataforma Qt. Este trabalho consiste na verificação de programas em C++ baseados no *framework* Qt, usando a ferramenta ESBMC++, a qual possui um *front-end* baseado em 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 os resultados associados 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++ [8, 10, 9, 26]. 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 aritméticow, 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, ou seja, não existe nenhuma possibilidade do usuário pré-processar programas em qualquer fase descrita [27, 28].

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++ [8], uma vez que a abordagem permite que o ESBMC++ encontre a maioria

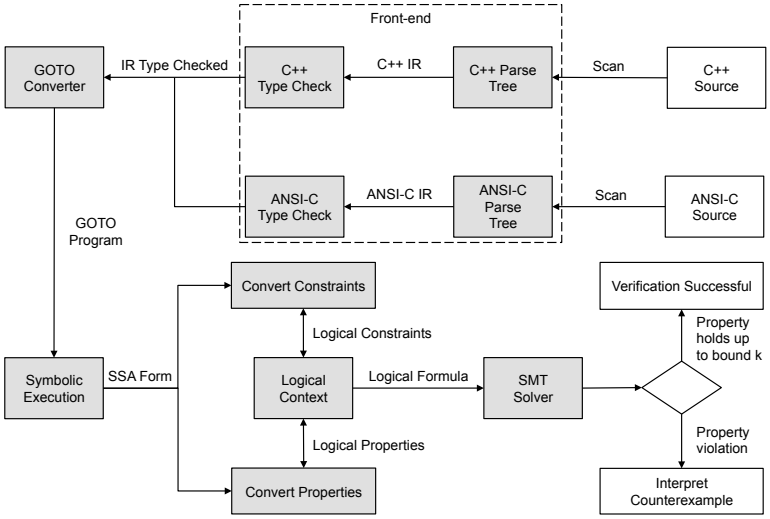


Figura 2.1: Visão geral da arquitetura do ESBMC++.

dos erros de sintaxe já relatados pelo GCC. Programas ANSI-C/C++/Qt são convertidos em 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 [8]. 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 por *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 *C*) e propriedades (fórmula *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++ [29], o qual atribui valores as 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 [30].

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 à funções não interpretadas, aritmética linear e não linear, vetores de bit, tuplas, vetores e outras teorias de primeira ordem. Dado uma teoria T e uma fórmula livre de quantificadores ψ, a respectiva fórmula, é satisfatível T se somente se existir uma estrutura que satisfaça tanto a fórmula quanto as sentenças de T, ou seja, se $T \cup \{\psi\}$ é satisfatível [31]. Dado um conjunto $\Gamma \cup \{\psi\}$ das fórmulas sobre T, ψ é uma consequência T de Γ ($\Gamma \models_T \psi$) se e somente se todo os modelos de $T \cup \Gamma$ é também um modelo de ψ. Desta forma, ao verificar $\Gamma \models_T \psi$ pode se reduzi-la para verificação de um satisfatível T de $\Gamma \cup \{\neg\psi\}$.

As teorias dos vetores dos solucinadores SMT são normalmente baseadas em axiomas de McCarthy [32]. 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, 25, 33]

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

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, solucianadores SMT lidam com termos baseados em suas teorias usando um procedimento de decisão [34].

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 [1] 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++ [1] e Java [35]. São fornecidos programas que são executados em diferentes plataformas tanto de hardware quanto de software com o mínimo de mundaças nas aplicações desenvolvidas com o objetivo de manter o mesmo desempenho. Samsung [36], Philips [37] e Panasonic [38] são alguma das empresas presentes na lista top 10 da Fortune 500 que utilizam Qt para o desenvolvimento de suas aplicações [1].

De acordo com o relatório do *Cross-Platform Tool Benchmarking* 2014 [39], 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* [1] é considerado um módulo base de Qt, pois, contêm 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 amplamentes conhecidos e usados em aplicações reais com Qt e consistem em um item muito importante nos processos de verificação.

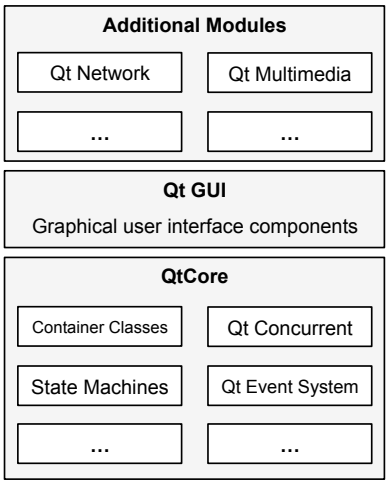


Figura 2.2: Visão geral da estrutura do *framework* Qt.

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* 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 disponivéis no 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 [40]. 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 [41].

Por fim e de acordo com o apresentado, nota-se que a complexidade e robustez de programas que utilizam o *framework* 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 como mostrado na 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.

Capítulo 3

SMT baseado em técnicas BMC para programas C++ que utilizam o *framework* Qt

Este capítulo descreve todo o processo de verificação com o ESBMC++; inicialmente o *parser* utilizado já havia sido mencionado na Seção 2.1 e nesta etapa é onde o ESBMC++ transforma o código de entrada em uma árvore IRep, a qual, possui todas as informações necessárias para o processo de verificação e, ao fim desta etapa, o ESBMC++ identifica cada estrutura presente no respectivo programa. No entanto, o ESBMC++ suporta apenas a verificação de programas ANSI-C/C++. Apesar dos códigos analisados do Qt serem escritos em C++, suas bibliotecas nativas possuem muitas estruturas hierárquicas e complexas. Desta maneira, o processo de verificação para essas bibliotecas e suas respectivas implementações otimizadas afetariam de forma desnecessária as VCs, além do mais poderiam não conter qualquer afirmação a cerca de propriedades específicas, tornando a verificação uma tarefa inviável.

O uso do QtOM proposto neste capítulo possui como objetivo solucinar o problema descrito acima, por ser uma representação simplificada, a qual considera a estrutura de cada biblioteca e suas respectivas classes associadas, incluindo atributos, assinaturas de métodos, protótipos de funções e *assertions*, garantindo assim 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, juntamente com pré e pós-condições que são necessárias para executar corre-

tamente os métodos do Qt. Vale ressaltar que QtOM é ligado de forma manual a ESBMC++, logo no início do processo de verificação, como mostrado na Figura 3.1. Desta forma, o QtOM pode auxiliar no processo de *parser* durante a criação de uma representação intermediária em C++ que possui todas os *assertions* indispensáveis para a verificação das propriedas mencionadas acima. Por fim, o fluxo de verificação segue de maneira tradicional.

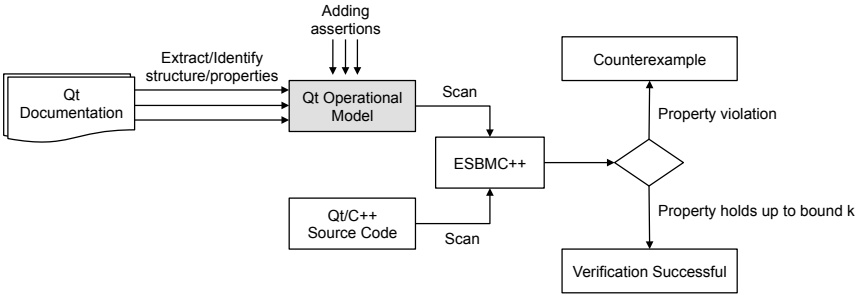


Figura 3.1: Conectando QtOM a ESBMC++.

Comparando com um trabalho anterior [11], o QtOM inclui atualmente uma representação inicial para todas as bibliotecas do módulo *QtCore* e *QtGui*, assim como, fornece um suporte completo a todas as classes *container* que são amplamente utilizadas em aplicações reais.

3.1 Pré-condições

Ao se utilizar modelos simplificados 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 *assertions* se torna indispensável para verificar as propriedades relacionadas aos métodos do framework Qt, assim como, as suas respectivas execuções que não são contempladas pelas bibliotecas padrões. Deste modo, tais *assertions* 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 *assertions*, ESBMC++ é capaz de verificar propriedas 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 ver-

ificar 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*, a qual fornece mensagens para aplicações e componentes do sistema na plataforma ArcGIS [23]. Como mostrado, o método *setFileName*() que manipula uma pré-condição (veja linha 6), aonde *m_inputFile* é um objeto da classe *QFile* que proporciona uma interface de leitura e escrita para se manipular arquivos [1]. Ao ser definido um nome ao arquivo, o objeto *fileName* de *QString* não pode ser nulo. Desta forma, quando *setFileName*() é chamado, ESBMC++ interpreta o seu comportamento de acordo com o implementado em QtOM.

```
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.

De acordo com a figura 3.3 que mostra um trecho do modelo operacional correspondente à classe QFile com uma implementação de *setFileName*() (veja as linhas 5-10), onde apenas os pré-requisitos são verificados. Em particular, se o objeto QString passado como um parâmetro não estiver vazio (veja a linha 6), a operação é válida e, consequentemente, a premissa estipulada é considerada *verdadeira*. Caso contrário, se uma operação errada for realizada, como uma string vazia for passada como parâmetro, a premissa estipulada é considerada *falsa*. Nesse caso, ESBMC++ retornaria um contra-exemplo com todos os passos necessários para reproduzir a execução de tal violação além de descrever o erro da respectiva premissa violada.

```
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.

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, o qual, em relação a corretude a cerca do valor impresso não foi abordado neste trabalho. Por fim, tais métodos só há suas 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 ser verificada.

3.2 Pós-condições

Em aplicações reais existem métodos que não contêm apenas propriedades que serão tratadas como pré-condição, mas tambem serão considerados pós-condição [11, 42]. Por exemplo, de acordo com a documentação do Qt [1], a função *setFileName* conforme descrita na Seção 3.1 não dever ser utilizada se o respectivo arquivo 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 instrução executada na linha 4 (veja Fig. 3.2) seria não determinística para o ESBMC++, assim como, a premissa presente na linha 8 do modelo operacional associado, como mostrado na Figura 3.3, uma vez que não há como se afirma 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 a manipulação de arquivos. Como a classe *QFile* indiretamente herda os métodos *open*() e *isOpen*(), a partir da classe *QIODevice*, para simulações comportamentais desses métodos constrói-se um modelo operacional para *QIODevice* conforme mostrado na Figura 3.4.

```
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*.

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* [1] com o objetivo de se obter o mesmo comportamento, assim como, incluir mecanismos necessários à 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 [43].

3.3 Resumo

Neste capítulo, foi descrito como é realizado o processo de verificação com ESBMC++ junto com o modelo operacional proposto (QtOM) de acordo como mostrado na figura 3.1, logo em seguida, é descrito como o modelo operacional proposto 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. Por fim, descreve-se como QtOM usa pós-condições para garantir o mesmo comportamento de métodos ou funções, a serem analisados, em relação ao apresentado nas bibliotecas originais do *framework*.

Capítulo 4

Modelo Operacional para containers

O módulo *Qt Core* possui um subconjunto de classes denominado *Qt Container* [1] como alternativa para os *containers* da *Standard Template Library* (STL) disponíveis na linguagem C++ [44]. 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.

Por fim, uma linguagem base é proposta na Seção 4.1 com o objetivo de formalizar a implementação de cada *container* e, em seguida, as Seções 4.2 e 4.3 descrevem as implementações formais para os *containers* sequenciais e associativos, respectivamente.

4.1 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.* [8] é utilizada e se

estende até a formulação das propriedades C e 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 4.1.

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

Figura 4.1: Sintaxe da linguagem adaptada, base para a descrição formal dos *conatiners*.

De acordo com a figura acima, os elementos básicos estão divididos em dois domínios sintáticos: V para valores e K para as chaves. 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 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.* [8].

4.2 Containers sequenciais

Containers sequenciais tem como objetivo armazenar elementos em uma determinada ordem [45]. De acordo com a documentação do Qt [1], *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 se utilizam da linguagem base descrita na seção 4.1. 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 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, $store(c_1, lower^1_{bound}, select(c_2, lower^2_{bound})) \ldots store(c_1,$ 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,

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

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.* [8].

4.3 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

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

onde $(k \to v)$ 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 [8]. Novamente, todas as operações dessas bibliotecas podem ser expressadas a partir de uma variação simplificada das três principais operações citadas na Seção 4.2.

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,

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

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,

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

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,

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

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. Na verdade, estas e outras propriedades específicas são tratadas em seus respectivos modelos operacionais conforme descrito no capítulo 3.

4.4 Resumo

Neste capítulo, foi descrito o subjconjunto denominado *Qt Container* que se faz presente no módulo *Qt Core* do *framework* multiplataforma Qt, sendo classificado em dois subgrupos: sequenciais e associativos, dependendo da estrutura de armazenamento implementada, logo em seguida, é descrito a linguaguem base formalizada por Ramalho *et al.* [8] 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 realizada.

Capítulo 5

Avaliação experimental

Este capítulo é dividido em três partes. Seção 5.1 descreve toda configuração experimental utilizada, os experimentos e todos os parâmetros de avaliação utilizados para a realização das avaliações. Na seção 5.2, a corretude e também o desempenho da metodologia proposta são verificados, utilizando programas Qt/C++ *single-thread* baseados na documentação do *framework* multiplataforma Qt [1]. Por fim, é descrito na seção 5.4 os resultados da verificação para as duas aplicações reais (Locomaps [22] e GeoMessage [23]) utilizando o modelo operacional proposto denominado QtOM.

5.1 Configuração Experimental

Com o intuito de avaliar a eficácia da abordagem proposta a cerca da verificação de programas que utilizam o *framework* Qt, um conjunto de testes automáticos denomidado *esbmc–qt* foi criado. Em resumo, neste conjunto de testes contém 711 programas Qt/C++ (12903 linhas de código), 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 (*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) que em sua maioria possuem acesso aos módulos *Qt Core* e *Qt GUI*. Alguns casos de teste foram desenvolvidos a partir da documenta-

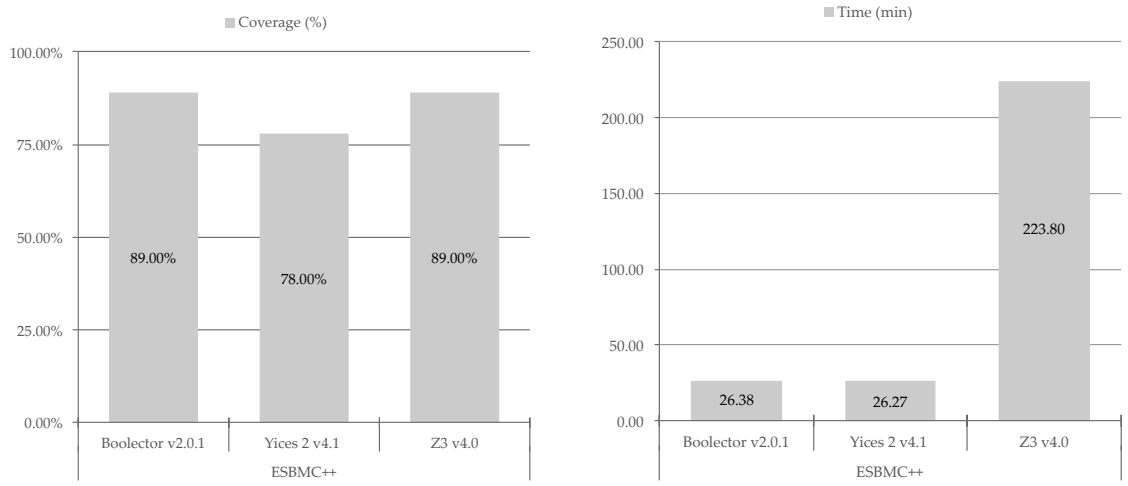
ção referente ao *framework* analisado e os restantes foram desenvolvidos especificamente para analisar todos as característica fornecidas pelo *framework*. Vale ressaltar que cada caso de teste é verificado manualmente antes de ser adicionado ao seu respectivo conjunto de teste. Desta forma, é capaz de se identificar se um determinado caso de teste possui ou não quaisquer erro e está de acordo com a operação a ser realizada. Assim, com base nesta revisão é possível garantir que 353 dos 711 casos de teste contêm erro, ou seja, 49.65% e 358 casos de teste não possuem falhas isto é 50.35%. Na realidade esse tipo de revisão é essencial para a nossa avaliação experimental uma vez que pode-se comparar os resultados obtidos através da verificação realizada pelas ferramentas utilizadas e avaliar adequadamente se erros reais foram encontrados.

Todos os experimentos foram realizados em um 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), executanto o sistema operacial denominado livre e chamado de Fedora de 64 bits se utilizando a ferramenta denominada ESBMC++ de versão 1.25.4 com três tipos de solucionadores instalados denominados, Z3 com versão 4.0, Boolector com versão 2.0.1 e Yices 2 com versão 4.1. Os limites de tempo e memória utilizados para cada caso de teste foram respectivamente definidos em 600 segundos e 22 GB. Em adição, uma avaliação foi realizada utilizando CBMC *v*5.1, LLBMC *v*2013.1 e DIVINE *v*3.3.2 combinados com o modelo operacinal proposto(QtOM) com o objetivo de proporcionar comparações entre ferramentas e em relação a ESBMC++. Os períodos de tempo foram indicados usando a função `clock_gettime` a partir da biblioteca `time.h` [46].

5.2 Comparação entre solucionadores SMT

É conhecido que diferentes solucionadores SMT podem afetar fortemente os resultados obtidos, uma vez que não existe homogeneidade em relação a abordagem de implementação e as lógicas suportadas. Primeiramente foram realizadas verificações usando os três solucionadores SMT mencionadas(Z3, Boolector e Yices). Sendo assim, Yices obteve os piores resultados, apresentando uma taxa de cobertura com 78% e um tempo de verificação com $26,27$ minutos. Por outro lado, tanto os solucionadores Z3 e Boolector apresentaram uma taxa de cobertura com 89% mas as verificações realizadas com Z3 se mostraram inferiores em relação as verificações feitas com Boolector a cerca do tempo de verificação, onde, respectivamente são apresentados tempos de verificação com $223,6$ minutos e $26,38$ minutos, sendo relatado quatro casos de teste com violação em relação ao tempo limite determinado, isto é, Boolector foi aproximadamente

8,5 vezes mais rápido que Z3 com a mesma precisão. Além disso, Yices apresentou a menor taxa de cobertura e tempo, pois, não possui suporte a tuplas. Desta forma, não conseguiu resolver corretamente as fórmulas SMT originadas do processo de verificação dos diversos casos de teste. Em resumo, de acordo com a figura 5.1 Boolector se apresenta como o melhor solucionador para o processo de verificação proposto.



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

Figura 5.1: Comparação entre solucionadores SMT.

5.3 Verificação dos resultados para o conjunto de casos de teste desenvolvidos

Todos os casos de teste presente na suite de teste esbmc-qt foram verificados de forma automática por ESBMC++ com o objetivo de analisar a sua corretude e eficiência. Além da comparação entre solucionadores SMT descrita acima, uma análise a cerca do desempenho entre ferramentas de verificação distintas também foi realizada. Como já mencionado não existe um verificador que análise o framework Qt e nem um modelo operacional semelhante ao proposto neste trabalho(QtOM) utilizando a linguagem C++. No entanto, devido à versatilidade de QtOM também é possível conectá-lo ao processo de verificação de LLBMC [21] e DIVINE [16], cuja base deste processo é a tradução código fonte em um representação intermediária denominada LLVM. Dessa forma, QtOM é usado como um apoio em seus processos de tradução, pois, o bitcode que logo em seguida é produzido contém informações a cerca

do código fonte utilizado na verificação e do modelo operacional proposto(QtOM). Por fim, foi feito uma comparação em relação ao desempenho de LLBMC e ESBMC++, que são verificadores baseados em técnicas SMT, e DIVINE, que emprega uma verificação de modelos através estados explícitos. Inicialmente, houve uma iniciativa de se realizar também uma comparação com CBMC [13] embora mesmo sendo utilizado o modelo operacional proposto não foi possível de realizar as verificações determinadas, isto já havia sido relatado em trabalhos anteriores por Ramalho et al. [8] e Merz et al. [21], o que ocasionou em sua remoção durante o processo de avaliação.

As ferramentas utilizadas foram executadas seguindo três roteiros. Um para ESBMC++ que identifica a partir de um arquivo seus parâmetros iniciais e realiza sua execução[1], outro para LLBMC que usando CLang[2] [17] compila o código fonte desejado criando seu *bitcode* e logo em seguida, também a partir de um arquivo identifica seus parâmetros iniciais e realiza a sua execução da ferramenta[3] e outro para DIVINE que também pré-compila os códigos fontes em C++ desejados criando seus respectivos *bitcode*[4] e em seguida realiza a verificação sobre eles[5]. O desdobramento de loops é definido para cada ferramenta, ou seja, o valor de <bound> mas este valor varia entre os casos de teste. Por enquanto, LLBMC não suporta tratamento de exceção e os *bitcodes* que foram criados sem exceção estavam a opção *-fno-exceptions* ativa em seu compilador, se está opção estiver ativa LLBMC sempre abortará durante seu processo de verificação.

A Tabela 5.1 mostra os resultados experimentais para as combinações entre QtOM e LLBMC, DIVINE e ESBMC++ usando Boolector como principal solucinador SMT. *CT* representa o número de programas que utilizam o framework Qt em 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, ou seja, resultados positivos corretos, *N* representa o número de casos de teste com defeitos, ou seja, resultados negativos correto, *FP* representa o número falsos positivos obtidos, ou seja, a ferramenta relata programas que estão corretos como incorretos, *FN* representa o número de falsos negativos obtidos, ou seja, a ferramenta relata pro-

[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

gramas incorretos como corretos e *Fail* representa o número de erros internos obtidos durante a verificação(*por exemplo*, erros de análise). Vale ressaltar que ESBMC++ utilizando Boolector não a estouro de memória e tempo em qualquer caso de teste utilizado.

			ESBMC++ v1.25.4						LLBMC v2013.1						DIVINE v3.3.2					
Testsuite	CT	L	Tempo	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 5.1: 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 acima, apenas $1,1\%$ dos casos de teste com ESBMC++ alegaram falhas durante sua verificação que ocorreu quando a ferramenta não foi capaz de realizar a verificação de um determinado programa devido a erros internos enontrados. DIVINE e LLBMC apresentam taxas de falhas a cerca de $6,9\%$ e $3,4\%$, respectivamente, quando tais ferramentas não conseguiram criar os *bitcodes* dos programas utilizados ou durante verificação relaizada foi relatado estouro de memória ou de tempo. Em relação aos resultados FP, DIVINE obteve o melhor desempenho seguido por LLBMC e ESBMC++. Contudo, ESBMC++ obteve a taxa mais baixa em relação aos resultados de falsos negativso(FN) seguido por DIVINE e por fim LLBMC, devido a forma que os iterators estão implementados no modelo operacional proposto(QtOM), através de ponteiros e vetores com o objetivo de simular seus comportamentos de forma real de acordo com o visto no capítulo 4. No entanto, a estrutura criada não cobre todos os comportamentos descritos na documentação do framework. Em particular, quando uma remoção de um elemento é realizada em um container em que existe mais de um iterator apontando para ele, todos os iterators que apontam para o elemento que foi removido serão perdidos. Desta forma, este comportamento afetará as pós-condições de um programa que

influenciam diretamente os resultados obtidos em relação a FP e FN. Vale ressaltar que os vetores e ponteiros têm sido extensivamente utilizados de modo a obter estruturas simples, isto é, sem classes e estruturas em sua representação o que diminui a complexidade do processo de verificação(ver seção 3.2). Por fim, a combinação entre os resultados de ESBMC++ e QtOM em um verificador robusto ainda possui algumas lacunas a serem preenchidas sobre o suporte da linguagem C++ como descrito por Ramalhoet al. [8].

Vale mencionar que o nível de complexidade ao se verificar o código fonte de um programa aumenta de acordo com a quantidade de linhas ele tiver, assim como, a quantidade de estruturas que possuir. No entanto, de acordo como mostrado na figura 5.2, os conjuntos de teste *QMap* e *QSet* apresentam os maiores tempos durante o processo de verificação ao se utilizar ESBMC++, apesar de *QVector* ser o mais extenso conjunto de casos de teste existente. Isso acontece devido não importar somente o número de linhas de código a ser analisado mas também a quantidade de loops presente no programa o que afeta diretamente os tempos de verificação. Na realidade, as estruturas internas do modelo operacional associadas a QMap e QSet contém mais loops do que as demais, desta forma, obtendo-se tempos de verificação mais longos. Como também visto na figura 5.2, LLBMC apresenta um maior tempo de verificação ao se utilizar o conjunto de teste *QVector*, na qual isto ocorre devido a dois casos de teste onde houve estouro do tempo estimado para que seja realizada a verificação. Além disso, DIVINE é a ferramenta que apresenta o menor desempenho entre as citadas, pois, seu processo de criação do *bitcode* 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 ao se utilizar DIVINE que no caso são QVector, QList, QMap, QLinkedList e QSet.

A figura 5.3 mostra todos os verificadores que 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 a partir 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 até $6,7\%$ para resultados falsos positivos e falsos negativos, ou seja, de 3 a 18 casos de teste dos 267 casos de teste devido as limitações relacionadas a 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 devido a falhas no processo de criação do *bitcode*. Em

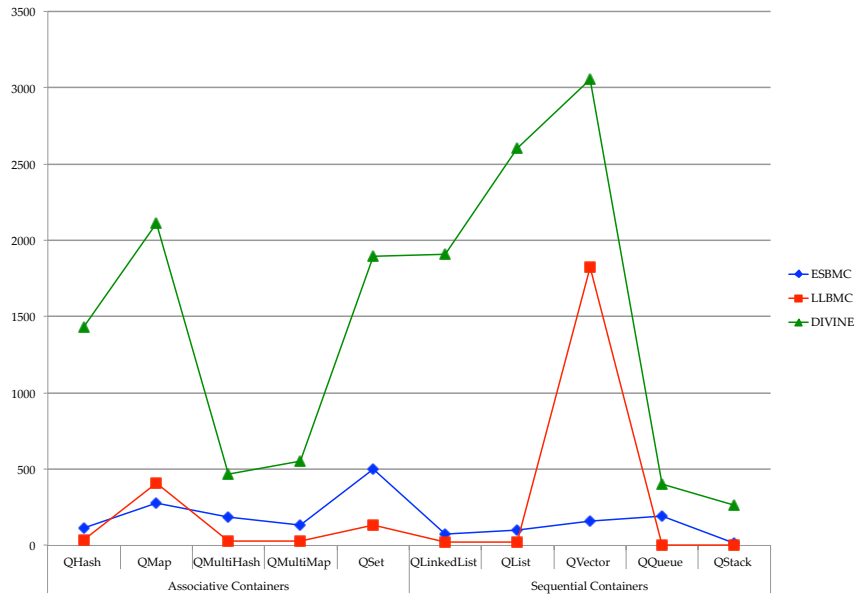


Figura 5.2: Comparação entre os tempos de verificação em relação a ESBMC++, LLBMC e DIVINE.

relação aos containers do tipo sequencial, LLBMC apresentou taxas de cobertura baixas para os conjuntos de teste QVector, QLinkedList e QList cerca de $67,7\%$ a 77%, ou seja, 84/117 de 124/152 dos casos de teste respectivamente, uma vez que cerca de $22,9\%$ dos casos de teste(83 dos 363 analisados) apresentaram resultados falsos negativos devido a também a problemas com a representação interna dos iterators. Além disso, cerca de 5% dos casos de teste(18 dos 363 analisados) não haviam sido verificados por LLBMC, uma vez que não foi capaz de criar os *bitcodes* desejados. 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 devido a 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 presente em QStack, pois, ao se utilizar o solucinador Boolector com a ferramenta ESBMC++ não foi possível obter-se uma solução para as fórmulas SMT criadas a partir deles para os casos analisados.

Os conjuntos de teste QList, QMap, QVector e QSet possuem mais resultados falsos positivos e negativos em seus teste ao se utilizar 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 de diferentes containers.

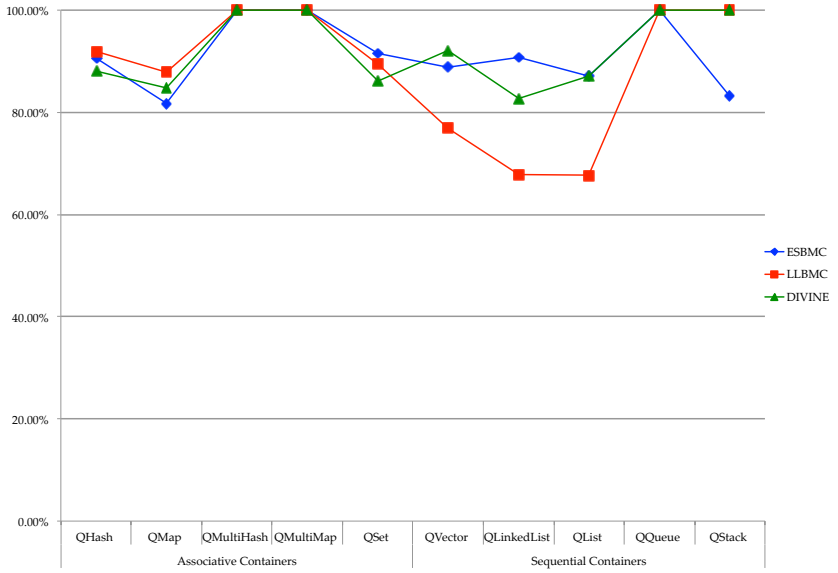


Figura 5.3: Comparação entre a taxa de cobertura em relação a ESBMC++, LLBMC e DIVINE.

Vale ressaltar que ESBMC++ foi capaz de identificar 89,3% dos erros nos casos de teste utilizados, ou seja, 631 dos 708 casos de testes utilizados possuiam erros o que demonstra também a sua eficácia. Similarmente, LLBMC e DIVINE apresentam, respectivamente, taxas com 81,4% e 89% ,isto é, 576 e 630 dos 708 casos de teste utilizados possuiam erros, isso também demonstra uma boa adequação do modelo operacional proposto(QtOM) combinado com outras ferramentas de verificação. Como consequência, a metodologia proposta não apenas se limita a uma determinada ferramenta, podendo-se adaptar para aplicações específicas em que algumas abordagens são mais adequedas do que outras.

5.4 Resultados da verificação para aplicações reais que utilizam o framework Qt

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 envolvem aplicações reais. Os parágrafos seguintes descrevem as respectivas aplicações e os resultados associados.

A aplicação chamada Locomaps [22] é um exemplo de programa que utiliza o framework Qt que exibi imagens de satélite, terrenos, mapas de ruas, serviço de planejamento *tiled map* e possui um integração com GPS Qt Geo. Utilizando o mesmo código fonte está apli-

cação 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 códigos 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 [47] para as forças armadas é um plataforma geográfica que é utilizada para criar, organizar e compartilhar materiais geográficos com usuário que utilizam mapas inteligentes online. A partir disso, GeoMessage Simulator [23] possui como entrada de dados arquivos XML e cria em diferentes frequências datagramas utilizando o protocolo de datagramas por usuário(*em ingles*, User Datagram Protocol(UDP)) para aplicações ArcGIS e componentes do sistema. GeoMessage 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, widgets e assim por diante. Vale ressaltar que GeoMessage usa duas classes, QMutex e QMutexLocker, relacionadas ao módulo Qt Threading que possui classes para programas concorrentes. Tais classes foram utilizados 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 proposto(QtOM) não fornece um suporte completo para o módulo Qt Threading ainda.

ESBMC++ junto ao modelo operacional proposto(QtOM) foi aplicado para verificar as aplicações Locomaps e GeoMessage buscando verificar as seguintes propriedades: violação dos limites de um array, aritméticas de under- e overflow, divisão por zero, segurança de ponteiro e outras propriedades específicas do framework definicas em QtOM de acordo com cápitulo 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, 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 a metodologia proposta levou aproximadamente 6.7 segundos para gerar 32 condições de verificação(*em inglês*, Verification Conditional(VC)) para Locomaps e 16 segundos para gerar 6421 condições de verificação para GeoMessage em um comum computador de mesa. Além disso, ESBMC++ não relata caso haja qualquer falso negativo mas foi ca-

paz 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 5.4: Fragmento de código do arquivo principal da aplicação Locomaps.

A figura 5.4 mostra um fragmento de código retirado do principal arquivo 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 linha2, na figura 5.4). A fim de verificar está propriedade, ESBMC++ analisa duas premissas em relação aos parâmetros de entrada da aplicação (veja as linhas 4 e 5, na figura 5.5), avaliando-as como pré-condições. Um erro semelhante também foi encontrado na aplicação GeoMessage e uma maneira possível para corrigir tal erro é sempre verificar, com instruções condicionais, se *argv* e *argc* são argumentos válidos antes de utilizalos 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 5.5: Modelo operacional para o construstor de *QApplication*().

5.5 Resumo

Resumo

Capítulo 6

Conclusões

A abordagem proposta neste trabalho tem como objetivo verificar programas que utilizam o *framework* Qt e foram desenvolvidos em C++/Qt, 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 também foi 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 [22] e outra que gera *datagramas broadcast* UDP com base em arquivos XML [23] foram verificadas usando o ESBMC++ com 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* Qt. Os experimentos que foram realizados envolvem programas em Qt/C++ com muitas características oferecidas pelas classes *container* do *framework* Qt.

Além disso, também foram avaliados o desempenho dos solucionadores Z3, Boolector e Yices, dado que eles foram utilizados no processo de verificação dos programas que utilizam o *framework* Qt com ESBMC++ e o modelo operacional proposto (QtOM). Como resultado, o Boolector apresentou a maior taxa de cobertura dos programas verificados e com um menor tempo de verificação.

Outra fundamental contribuição é a integração de QtOM dentro do processo de verifi-

cação de outro dois diferentes verificadores de modelo conhecidos como LLBMC e DIVINE, demonstrando a flexibilidade 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 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 reias que estão escritos em C++/Qt para cenários específicos e aplicações.

6.1 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* Qt. Além disso, mais classes e biblitecas serão adicionadas com o intuito de aumentar a cobertura da verificação em relação ao *framework* Qt e dessa forma validar suas respecitvas propriedades. Por fim ferramentas para medição de desempenho do modelo operacional proposto serão incluidas, a fim de verificar se rotinas específicas estão de acordo com as limitações de tempo.

Incluir um cronograma com as tarefas remanescentes.

Referências Bibliográficas

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

[2] BERARD, B. et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642074782, 9783642074783.

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

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

[5] MEHLITZ, P.; RUNGTA, N.; VISSER, W. A hands-on java pathfinder tutorial. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 1493–1495. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2487052>.

[6] MERWE, H. van der; MERWE, B. van der; VISSER, W. Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes*, v. 39, n. 1, p. 1–5, 2014. Disponível em: <http://doi.acm.org/10.1145/2557833.2560576>.

[7] MERWE, H. van der et al. Generation of library models for verification of android applications. *ACM SIGSOFT Software Engineering Notes*, v. 40, n. 1, p. 1–5, 2015. Disponível em: <http://doi.acm.org/10.1145/2693208.2693247>.

[8] RAMALHO, M. et al. 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. Washington, DC, USA: IEEE Computer Society, 2013. (ECBS '13), p. 147–156. ISBN 978-0-7695-4991-0. Disponível em: <http://dx.doi.org/10.1109/ECBS.2013.15>.

[9] 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*. [s.n.], 2011. p. 331–340. Disponível em: <http://doi.acm.org/10.1145/1985793.1985839>.

[10] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, v. 38, n. 4, p. 957–974, 2012.

[11] 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*. [S.l.]: IEEE, 2015.

[12] FALKE, S.; MERZ, F.; SINZ, C. The bounded model checker LLBMC. In: *ASE*. [s.n.], 2013. p. 706–709. Disponível em: <http://dx.doi.org/10.1109/ASE.2013.6693138>.

[13] KROENING, D.; TAUTSCHNIG, M. CBMC - C bounded model checker - (competition contribution). In: *TACAS*. [s.n.], 2014. (LNCS, v. 8413), p. 389–391. Disponível em: <http://dx.doi.org/10.1007/978-3-642-54862-$8_2$6>.

[14] WANG, W.; BARRETT, C.; WIES, T. Cascade 2.0. In: *VMCAI*. [s.n.], 2014. (LNCS, v. 8318), p. 142–160. Disponível em: <http://dx.doi.org/10.1007/978-3-642-54013-4_9>.

[15] MOURA, L. M. de; BJØRNER, N. Z3: an efficient SMT solver. In: *TACAS*. [s.n.], 2008. (LNCS, v. 4963), p. 337–340. Disponível em: <http://dx.doi.org/10.1007/978-3-540-78800-$3_2$4>.

[16] BARNAT, J. et al. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: *Computer Aided Verification (CAV 2013)*. [S.l.]: Springer, 2013. (LNCS, v. 8044), p. 863–868.

[17] LATTNER, C. *CLang Documentation*. [S.l.], 2015. [Online; accessed December-2015].

[18] BLANC, N.; GROCE, A.; KROENING, D. Verifying C++ with STL containers via predicate abstraction. In: *ASE*. [s.n.], 2007. p. 521–524. Disponível em: <http://doi.acm.org/10.1145/1321631.1321724>.

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

[20] SITES, R. L. *Some Thoughts on Proving Clean Termination of Programs.* Stanford, CA, USA, 1974.

[21] MERZ, F.; FALKE, S.; SINZ, C. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: *VSTTE*. [s.n.], 2012. (LNCS, v. 7152), p. 146–161. Disponível em: <http://dx.doi.org/10.1007/978-3-642-27705-4_12>.

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

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

[24] DUTERTRE, B. Yices 2.2. In: BIERE, A.; BLOEM, R. (Ed.). *Computer-Aided Verification (CAV'2014)*. [S.l.]: Springer, 2014. (Lecture Notes in Computer Science, v. 8559), p. 737–744.

[25] BRUMMAYER, R.; BIERE, A. Boolector: An efficient SMT solver for bit-vectors and arrays. In: *TACAS*. [S.l.: s.n.], 2009. (LNCS, v. 5505), p. 174–177.

[26] MORSE, J. et al. Model checking ltl properties over ansi-c programs with bounded traces. v. 14, p. 65–81, 2013. ISSN 1619-1374. Disponível em: <http://dx.doi.org/10.1007/s10270-013-0366-0>.

[27] MORSE, J. et al. Tools and algorithms for the construction and analysis of systems. In: ÁBRAHÁM, E.; HAVELUND, K. (Ed.). [S.l.]: Springer Berlin Heidelberg, 2014. cap. ESBMC 1.22, p. 405–407. ISBN 978-3-642-54862-8.

[28] MORSE, J. et al. Tools and algorithms for the construction and analysis of systems. In: PITERMAN, N.; SMOLKA, S. A. (Ed.). [S.l.]: Springer Berlin Heidelberg, 2013. cap. Handling Unbounded Loops with ESBMC 1.20, p. 619–622. ISBN 978-3-642-36742-7.

[29] CORDEIRO, L. et al. Tools and algorithms for the construction and analysis of systems. In: FLANAGAN, C.; KÖNIG, B. (Ed.). [S.l.]: Springer Berlin Heidelberg, 2012. cap. Context-Bounded Model Checking with ESBMC 1.17, p. 534–537. ISBN 978-3-642-28756-5.

[30] ROCHA, H. et al. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: *IFM*. [s.n.], 2012. (LNCS, v. 7321), p. 128–142. Disponível em: <http://dx.doi.org/10.1007/978-3-642-30729-4$_1$0>.

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

[32] MCCARTHY, J. Towards a mathematical science of computation. In: *In IFIP Congress*. [S.l.]: North-Holland, 1962. p. 21–28.

[33] BARRETT, C.; TINELLI, C. CVC3. In: *CAV*. [S.l.: s.n.], 2007. (LNCS, v. 4590), p. 298–302.

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

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

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

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

[38] 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].

[39] RESEARCH2GUIDANCE. *Cross-Platform Tool Benchmarking*. [S.l.], 2014.

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

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

[42] MUSUVATHI, M. et al. Cmc: A pragmatic approach to model checking real code. ACM, New York, NY, USA, v. 36, p. 75–88, 2002. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/844128.844136>.

[43] PEREIRA, P. A. et al. Verifying cuda programs using smt-based context-bounded model checking. 2016.

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

[45] DEITEL, P.; DEITEL, H. *C++ How to Program*. [S.l.]: Prentice Hall, 2013. 1080 p. ISBN 978-85-7605-056-8.

[46] 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].

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