Capítulo 1

Introdução

A atual crescente disseminação a respeito de 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 tem crescido em relação a sua robustez e complexidade, onde se torna visível o uso de processadores com vários núcleos, memórias compartilhadas escalonavéis entre outros avançados recursos, de maneira à suprir o crescimento do poder computacional exigido, na qual, pode-se obter através da combinação de linguagens de programação e frameworks. Neste contexto, surgi o Qt como um poderoso framework multi-plataforma para dispostivos que enfatiza a criação de interface para usuário (IU) e o desenvolvimento de aplicações gráficas [1]. Contudo, como a complexidade de tais sistemas tende a crescer, desta forma, 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 eletrônicas de consumo cada vez mais investem em tempo e esforço para desenvolver alternativas rápidas e baratas de verificação, com o objetivo de verificar a corretude de seus sistemas, logo, evitar perdas fincanceiras [2]. Entre as alternativas já existente, uma das mais eficiente e menos custosa é a abordagem da verificação através de modelos [3]. No entanto, existem muitos sistemas que não se pode verifica-lós de uma maneira automática devido à incapacidade de alguns verificadores em suportar certos tipos de linguagens e frameworks. Por exemplo, Java PathFinder é capaz de verificar códigos em Java baseado em byte-code [4], mas não há suporte a verificação(completa) de aplicações Java que utilizam o sistema operacional

Android [5]. Na realidade, está verificação somente se tornará possível ao menos que haja uma representação abstrata das bibliotecas associadas, denominado modelo operacional(MO), que de forma conservadora aproxima-se a semântica usada pelo sistema operacional [6].

Este trabalho identifica as principais características do framework Qt e propõe um modelo operacional(MO) que prover analisar e verificar as propriedades relacionadas de acordo com as suas funcionalidades. Os algoritmos desenvolvidos foram intregrados em uma ferramenta bounded model checking(BMC) baseada em satisfiability modulo theories(SMT), denominada Efficient SMT-based Context-Bounded Model Checker(ESBMC++) [7], 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 programa em C++ como descrito por Ramalho *et al*. [7]. No entanto, na metodologia proposta, um MO é utilizado para identificar elementos do framework Qt e verificar propriedades específicas relacionadas a essas estruturas através de pré e pós-condições.

1.1 Descrição do Problema

Descrição

1.2 Objetivos

Objetivos

1.3 Contribuições

O trabalho é uma extensão de trabalhos já publicados anteriormente [8]. O respectivo MO foi ampliado com o objetivo de incluir novas funcionalidades de principais módulos do Qt, neste caso, QtGui e QtCore. Em verdade, as principais contribuições aqui são:

- Suportar containers baseados em templates sequenciais e associativos;
- Integrar QtOM ao processo de verificação de programas em C++ em verificadores que se encontram no estado da arte, neste caso, DIVINE [9] e LLBMC [10];

- Criar o suporte a verificação de duas aplicações baseadas em Qt denominadas Locomaps [11] e Geomessage [12], respectivamente.

- Avaliar o desempenho de três solucionadores SMT(Z3 [13], Yices [14] e Boolector [15]) sobre o conjunto benchmark utilizado extensivamente e ampliado em relação ao trabalho anterior juntamente com a abordagem proposta.

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

1.4 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 eletrônicas de consumo que se utilizam deste framework, apresentam diversas propriedades que devem ser verificadas como, aritmética under- e overflow, segurança de ponteiros, limite de vetores, corretude no uso de containers, metódos de Qt Framework, chamadas de função, escalonamento de ventos, entre outros. Além disso, a verificação através de testes manuais é um processo árduo e custoso. Em resumo, a técnica BMC aplicada em verificação de software é usada em diversos verificadores [10, 16, 17, 18] e está se tornando popular, principalmente, devido ao aumento de solucionadores SMT cada vez mais sofisticados que são baseados em solucionadores efficient boolean satisfiability(SAT) [13].

Merz, Falke e Sinz [10] apresenta o Low-Level Bounded Model Checker (LLBMC) como um verificador que utiliza modelos operacionais para verificar códigos 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, desta forma, executar as verificações propostas. De forma semelhante ESBMC++, Merz, Falke e Sinz também utilizam solucionadores SMT para analisar as condições de verificação. Contudo, diferente da abordagem aqui proposta, LLBMC não suporta tratamento de exceção o que acarreta numa verificação de programas reais escritos em C++ menos custoso, por exemplo, programas baseados em Standard Template Libraries - STL.

Vale ressaltar que a representação intermediária usado por LLVM perde algumas informações sobre a estrutura dos respectivos programas em C++ , ou seja, as relações entre classes.

Barnat *et al.* apresenta DIVINE [9] como um verificador de modelo de estado explícito para programas ANSI-C/C++ simples e multithreads 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 [19] como front-end, a fim de converter programas em C++ à uma representação intermediária utilizada por LLVM que logo em seguida realiza a verificação sobre o bitcode criado. Embora DIVINE possua uma implementação de ANSI-C e das bibliotecas padrões do C++ o que lhe permite realizar uma verificação apropriada sobre programas desenvolvidos com essas linguaguens não possui qualquer suporte a representação disponibilizada pelo framework Qt. De acordo com a abordagem proposta, DIVINE é capaz de criar um programa que possa ser interpretado totalmente por 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 predicada [20], com o uso de dados de tipo abstrato sendo usados para realizar a verificação de STL ao invés de usar a implementação e o comportamento atual de componentes STL. Na verdade, eles 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 (visualizar Seção 3.1).

O C bounded model checker (CBMC) implementa a técnica BMC para programas ANSI-C/C++ através de solucionadores SAT/SMT [16]. Vale ressaltar que ESBMC foi construído em cima do CBMC, portanto, eles possuem processos de verificação semelhantes. Na verdade, CBMC pode processar programas usando a ferramenta goto-cc [21] que compila o código fonte da aplicação para um GOTO-programs equivalente (ou seja, um gráfico do controle de fluxo), a partir de um modelo compátivel ao GCC. A partir de GOTO-programs CBMC cria uma árvore abstrata de sintaxe (AST) que é convertida em um formato independente da linguaguem interna usada para as etapas restantes. CBMC também utiliza duas funções recursivas C e P que reg-

istram 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 a aritmética de under e overflow, violação no limite dos vetores e checagem de ponteiro nulo [22]. Por fim, um gerador de condições de verificação (VCG), em seguida, criar condições de verificação (VCs) a partir das fórmulas criadas e os envia para um solucionador SAT/SMT. Embora CBMC esteje relacionado como susposto verificador de programas em C++, Ramalho *et al.* [7] e Merz *et al.* [23] relatam que falhou ao verificar diversas aplicações simples em C++, o que também foi confirmado neste trabalho (visualizar Seção 5.2.1).

Por fim, destaca-se que QtOM está completamente escrito em C++, o que facilita a integração dentro de processos de verificação de outros verificadores. No presente trabalho, QtOM não foi somente intregado a ESBMC++ mas também em DIVINE [9] e LLBMC [10], afim de obter uma avaliação mais justa sobre a abordagem proposta.

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 juntar referências sobre o tema. Este trabalho está organizado da seguinte forma:

- O Capítulo 2, por sua vez, apresenta uma breve introdução sobre a arquitetura de ESBMC++ e satisfiability modulo theories(SMT) e também 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 mostra imagens de satélites, terrenos, mapas de ruas e Tiled Map Service(TMS) panning entre

outras características [11] 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

Neste capítulo, a arquitetura de ESBMC++ é descrita, assim como, é realizado uma descrição de algumas características estruturais do framework multiplataforma Qt. A maior concentração deste 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 VCs a um solucionador SAT, ESBMC++ os codifica através de diferentes teorias SMT e em seguida, passa os resultados associados a um solucinador SMT.

2.1 ESBMC++

ESBMC++, cujo possui a arquitetura mostrado na figura 2.1, é um context-bounded model checker baseado em solucionadores SMT usado em programas ANSI-C/C++ [7, 17, 24]. ESBMC++ verifica programas simples e multi-threaded e analisa propriedades relacionadas à aritmética de under- e overflow, divisão por zero, indíces fora do limite, segurança de ponteiros, deadlocks e corrida de dados. Em ESBMC++, o processo de verificação se encontra totalmente automatizado, 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.

No momento da verificação, primeiramente é realizado o parser do código fonte a ser analisado. Na verdade, o parser utilizado em ESBMC++ é fortemente baseado no compilador GNU C++, uma vez que a abordagem permite que ESBMC++ encontre a maioria dos erros de

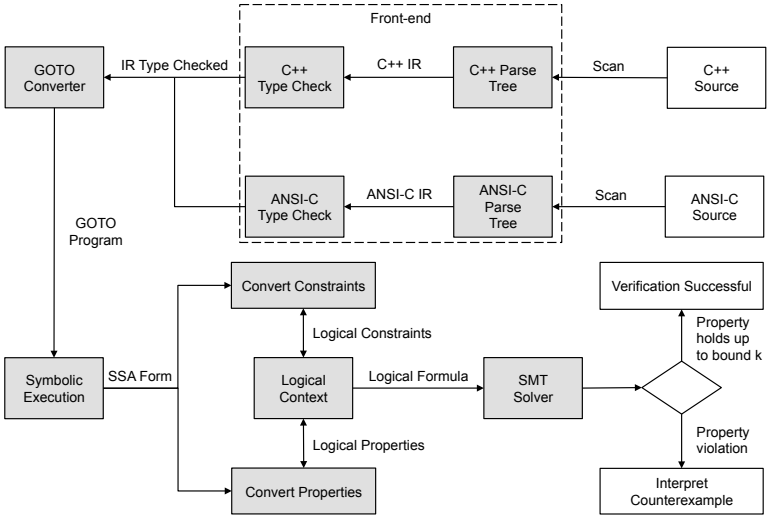


Figura 2.1: Visão geral da arquitetura de ESBMC++.

sintaxe já relatados por GCC. Programas ANSI-C/C++/Qt são convertidos em uma árvore de representação intermédiaria (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 Cáp. **??**.

Na etapa seguinte, denominada type-check, 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 [7]. 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 Single Static Assignment(SSA) é criada. Baseado nisso, ESBMC++ cria duas formulas 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 controle de fluxo 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 por ESBMC++, o qual, atribui valores a 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.

2.2 Satisfiability Modulo Theories (SMT)

SMT determina a satisfatibilidade das fórmulas de primeira ordem, usando uma combinação de suas teorias a fim de generalizar a satisfabilidade proposicional, dando suporte a funções não interpretadas, aritmética linear e não linear, bit-vectors, tuplas, vetores e outras teorias de primeira ordem de decisão. 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 [25]. 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 de solucinadores SMT são normalmente baseadas em axiomas de McCarthy [26]. 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 exatament o mesmo que a ao menos que o valor do índice i seja v. Formalmente,*select* e *store* pode então ser caracterizados por axiomas [13, 15, 27]

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

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 em ANSI-C, struct e 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 [28].

2.3 O framework multiplataforma Qt

Diversos módulos de softwares, conhecidos como frameworks, tem sido utilizados para acelerar os processos de desenvolvimento de aplicações. Diante desse contexto, o framework multiplataforma Qt [1] representa um grande 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 [29]. 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 [30], Philips [31] e Panasonic [32] 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 Cross-Platform Tool Benchmarking 2014 [33], 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 na figura 2.2. O módulo QtCore [1] 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 amplamentes conhecinhos e aplicados em aplicações reais com Qt e consistem em um item muito importante nos processos de verificação.

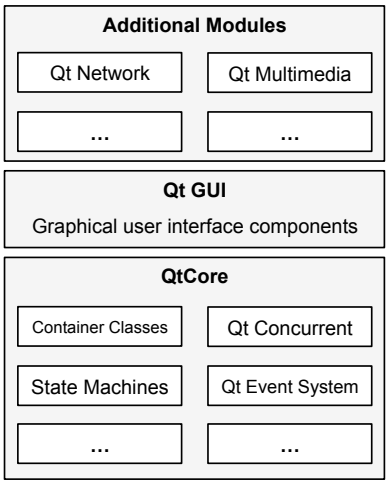


Figura 2.2: Visão geral da estrutura do framework Qt.

Além desses submódulos, QtCore também contém um sistema de eventos denominado Qt Event, onde, Qt representa um evento através de um objeto que herda a classe QEvent(classe

base para o sistema Qt Event) na qual contém informações necessárias sobre todas as ações(internas ou externas) relacionadas a uma 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 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 [34]. 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 [35].

Por fim e de acordo com o apresentado, nota-se que a complexidade e robustez de programas que se utilizam o framework Qt afeta diretamente os processos de verificação relacionados a eles. Em resumo, 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

Resumo

Capítulo 3

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

Neste capítulo, é descrito 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 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 ESBMC++ identifica cada estrutura presente no respectivo programa. No entanto, ESBMC++ suporta apenas a verificação de programas ANSI-C/C++ apesar de os códigos analisados do Qt serem escritos em C++ mas suas bibliotecas nativas possuem muitas estruturas hierárquicas e complexas. Desta maneira, o processo de verificação para essas bibliotecas e suas respectivas implementações otimizadas iriam afetar de forma desnecessária os VCs e além do mais poderiam não conter qualquer afirmação a cerca de propriedades específicas tornando a verificação uma tarefa inviável.

O uso de QtOm proposta neste capítulo possui o objetivo de solucinar o problema descrito acima por ser uma representação simplificada, a qual, considera a estrutura de cada biblioteca e suas respectivas classes associadas, incluindo assim atributos, assinaturas de métodos, protótipos de funções e premissas, garantindo assim que cada propriedade seja formalmente verificada. Na verdade, existem muitas propriedades a serem verificadas como acesso de memória inválida, valores negativos de tempo de período, acesso a arquivos inexistentes e ponteiros nulos, juntamente com pré e pós-condições que são necessárias para executar corretamente os

métodos de 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, QtOM pode auxiliar o processo do parser na criação de uma representação intermediária em C++ que possui todas as premissas indispensáveis para a verificação das propriedas acima mencionadas. Por fim, o fluxo de verificação segue de maneira tradicional.

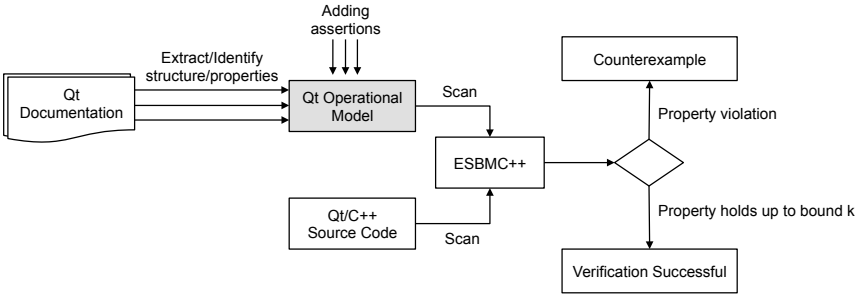


Figura 3.1: Conectando QtOM a ESBMC++.

Comparando com um trabalho anterior [8], 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 associado às árvores IRep. Desta forma, 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 premissas 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 premissas são integradas aos respectivos métodos com o objetivo de detectar violações de acordo com o uso incorreto do framework Qt.

Em resumo, baseado na adição de premissas, 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.

Conforme o fragmento de código mostrado na figura 3.2 presente na aplicação chamada GeoMessage Simulator, o qual, fornece mensagens para aplicações e componentes do sistema na plataforma ArcGIS [12]. 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 há métodos que não contêm apenas propriedades que serão manuseadas como pré-condição mas tambem serão considerados pós-condição [8]. Por exemplo, de acorodo com a documentação do Qt [1], a função *setFileName* conforme descrita na seção3.1 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 simulador 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 construi-se um modelo operacional para QIODevice como 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 dever ser mantida a fim de permitir uma adequada execução de futuras instruções, deve 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 são uma cópia simplificada das bibliotecas originais como todos os mecanismos necessários para a verificação do código.

3.3 Resumo

Resumo

Capítulo 4

Modelo Operacional para containers

O módulo Qt Core possui um template base de classes container como alternativa para containers STL usado na linguaguem C++ [1]. Por exemplo, durante desenvolvimento de uma aplicação é necessário a criação de uma pilha de tamanho variável usando QWidgets, uma alternativa seria o uso de QStack<QWidget>. Além disso, estes containers também usam iterators Java- e STL-style, de modo a se deslocar ao longos dos dados armazenados na estrutura criada.

Desta forma, esses tipos de classes podem ser classificados em dois subgrupos: sequenciais e associativas, dependendo da estrutura de armazenamento desenvolvida. Classes QList, QLinkedList, QVector, QStack e QQueue são classificadas como estruturas sequenciais, enquanto as classes QMap, QMultiMap, QHash, QMultiHash e QSet pertencem ao grupo das estruturas associativas.

Por fim, de acordo com a seção 4.1 uma linguagem principal é desenvolvida com o objetivo de formalizar a implementação de cada classe container, e em seguida, as seções 4.2 e 4.3 descrevem as implementações formais para os containers sequenciais e associativos.

4.1 Linguagem

Com o intuito de implementar o modelo operacional para Qt containers, a formalização da linguagem para o container core descrita por Ramalho*et al.* [7] é utilizada e estende-se até a formulação das propriedades *C* e *P*. Desta forma, a linguagem core é adaptada com o objetivo de formular adequadamente a verificação de ambos containers de Qt seja ele 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 de container Core para QtOM.

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 e v, do tipo K e V são adicionados respectivamente. Dessa forma, a notação I_v representa um valor armazendo em um container de base em uma posição direcionada por um iterator I e I_k representa uma chave. 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 uma tupla t que no campo f possui um valor v. Da mesma forma que P_k e P_v representam chave e valor em uma posição P respectivamente.

Entretanto, três outros métodos foram incluídos definidos como, $C.insert(k, v)$, o qual, insere um elementro numa estrutura container, com uma chave k e um valor v, possui como retorno um iterator que aponta para o novo elemento cuja posição depende do tipo de container usado, $C.search(k)$, o qual, 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 mas com um valor k correspondente. No entanto, ambos os métodos se não existir nenhuma chave ou valor correspondente é retornado $C.end()$. 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.

Todos os elementos restantes, a partir da linguaguem Core menncionada, são usadas aqui, de acordo como descrito por Ramalho *et al.* [7].

4.2 Containers sequenciais

Containers sequenciais em Qt são criados em uma estrutura, qual tem como objetivo armazenar elemento e uma determinada ordem sequencial [36]. De acordo com a documentação do Qt, QList é a classe container frequentemente mais utilizada e possui uma estrutura em formato de lista, a fim de armazenar os valores que podem ser acessados através de um índice. Da mesma forma, QLinkedList também possui um estrutura em forma de lista embora seja acessada atráves de iterators ao invés de índices inteiros. Na classe QVector há presente uma estrutura de array redimensionável e por fim, QStack e QQueue fornecem estruturas que implementam diretivas como o último a entra é o primeiro a sair(em inglês, *last in first in*) e o primeiro a entrar é último a sair(em inglês,*first in first out*), respectivamente.

Para simular adequadamente os containers sequenciais, os modelos propostos se utilizam da linguagem core a qual foi 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 container (onde $C_{size} \in \mathbb{N}$). Dessa forma, os iterators são modelados por meio de duas variáveis, uma do tipo inteiro que é denominado de i_{pos} e contém o valor do índice apontado por um iterator e uma 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 $c.search(v)$ considerando-se realizar uma pesquisa por um elemento v no container desenvolvido. Então, se esse elemento for encontrado, é retornado um iterator que aponta para o respectivo elemento, caso contrário, é retornado um iterator que imediatamente aponta para a posição após o último elemento do container(isto é, $c.end()$). Desta forma, a instrução "$c.search(v)$;" torna-se "$(c',i') = c.search(v)$;" que possuem efeitos adverso de forma explícita. Assim, a função de tradução C descreve premissas que estão relacionadas com o "antes"(em inglês, *before*) e o "depois"(em inglês, *after*) das respectivas versões das variáveis do modelo. Na verdade, notações com apóstrofo (por exemplo, c' and i') representam o estado das variáveis do modelo após realizar a execução da respectiva operação e notações simplificadas (por exemplo, c and i) representam os estados anteriores. Além

disso, $select(c, i = lower_{bound} \ldots i = upper_{bound})$ representa uma expressão de loop (como, for e $while$) onde cada valor de c, a partir de posições $lower_{bound}$ a $upper_{bound}$, 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 posições $lower^2_{bound}$ a $upper^2_{bound}$, serão armazenados em c_1 nas posições $lower^1_{bound}$ a $upper^1_{bound}$, respectivamente. Sendo assim,

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

Em relação aos containers sequenciais, os métodos $C.insert(I, V, \mathbb{N})$ e $C.erase(I)$ se comportam como descrito por Ramalhoet al. [7].

4.3 Containers Associativos

O grupo dos containers associativos possuem cinco classes: QMap, QMultimap, QHash, QMultiHash e QSet. QMap tem como abordagem um array associativo que conecta cada umas das chaves, de um certo tipo K, para um valor de um certo tipo V, onde as chaves associadas são armazenadas em ordem. Por um lado, QHash apresenta um comportamento ao de QMap, contudo, os dados são armazenados numa ordem arbitrária. QMultiMap e QMultihash representam respectivamente subclasses de QMap e QHash, ainda assim, ambas classes especificam interfacer aonde uma chave pode ser associada a diversos valores. Por fim, um valor único a partir de um conjunto matemático é a abordagem definida por QSet.

Com o intuito de implementar containers associativos, um ponteiro c_v é definido para os valores armazenados no container a ser desenvolvido e um ponteiro c_{size} também é usado para guardar o tamanho do container. No entanto, um ponteiro c_k é utilizado para a chave do container. Em particular, c_k e c_v estão conectados através 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

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

aonde $(k \to v)$ 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 [7]. Novamente, todas as operações dessas bibliotecas podem ser expressadas a partir de uma variação simplificada dos três principais como citado 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 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, o valor v associado com a 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. Desta forma, todas as variáveis acima referidas são considerados e uma comparação é realizada,

a fim de assegurar que o novo elemento é inserido da 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 para verificar se o elemento já existe com uma respectiva chave é ignorada. Por fim, com o propósito de realizar uma exclusão o método apagar(*em inglês*, erase) é criado, o qual é respresentado 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 criados induzem implicitamente duas principais propriedades com o objetivo de executar de forma correta as respectivas operações. A princípio c_k e c_v são considerados não vazios, ou seja, c_{size} também não é nulo para as operações de busca e exclusão de elementos. Por outro lado, i é considerado como um iterator sobre o respectivo container subjacente, isto é, dado um container c com os pointeiros 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

Resumo

Capítulo 5

Avaliação experimental

Este capítulo é dividido em três partes. Seção 5.1 descreve a toda configuração, os experimentos e todos os parâmetros de avaliação utilizados para a realização das avaliações. Na seção 5.2, a corretude e também o desempenho do método proposto são verificados utilizando programas C++/Qt utilizando uma única thread e que em sua maioria foram retirados da documentação do Qt [1]. Por fim, é descrito os resultados da verificação para as duas aplicações reais(Locomaps [11] e GeoMessage [12]) através do modelo operacional proposto denominado QtOM na seção **??**.

5.1 Configuração Experimental

Como 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 está contido 711 programas Qt/C++(12903 linhas de código), ou seja, todos os casos de teste utilizadas na atual avaliação.

Os casos de teste referidos acima estão divididos em 10 principais conjuntos de teste. Denominados QHash, QLinkedList, QList, QMap, QMultiHash, QMultiMap, QQueue, QSet, QStack e QVector com casos de teste para as respectivas classes container que em sua maioria tem acesso aos módulos Qt Core e Qt GUI. Alguns desses casos de teste foram diretamente retirados da documentação sobre o Qt e os restantes foram desenvolvidos especificamente para testar todos os recursos fornecidos pelo framework Qt. Vale ressaltar que cada caso de teste é verificado manualmente antes de ser adicionado ao seu respectivo conjunto de teste. Desta

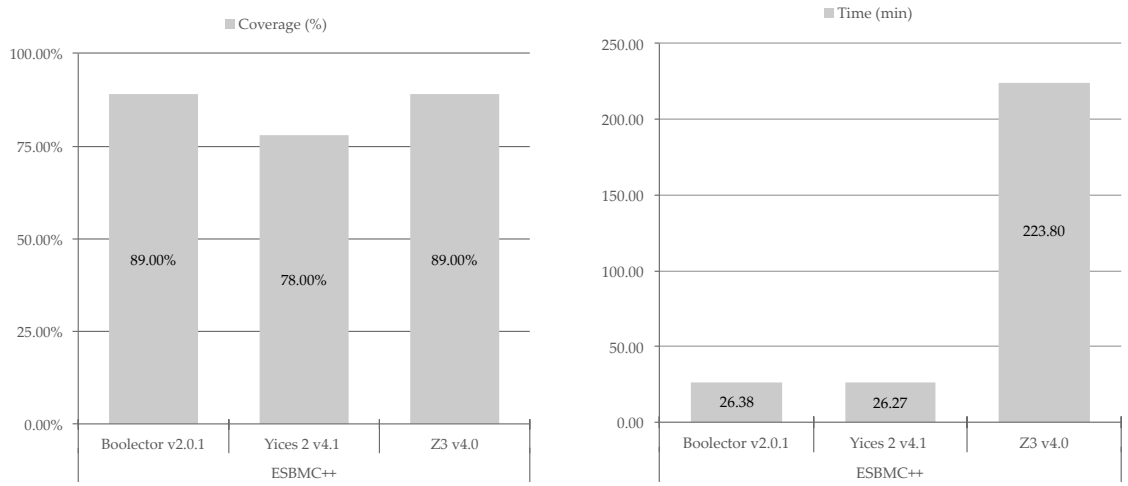
forma, é capaz de se identificar se um determinado caso de teste possui ou não quaisquer erro e está de acordo com a operação a ser realizada. Assim, com base nesta revisão é possível garantir que 353 dos 711 casos de teste contêm erro, ou seja, 49.65% e 358 casos de teste não possuem falhas isto é50.35%. 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` [37].

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.

5.2.1 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 [23] e DIVINE [9], 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 [16]

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. [7] e Merz et al. [23], 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] CLANG 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 programas 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.

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 re-

[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

			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.

alizar 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 FN seguido por DIVINE e 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(Ver seção **??**). 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. [7].

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.

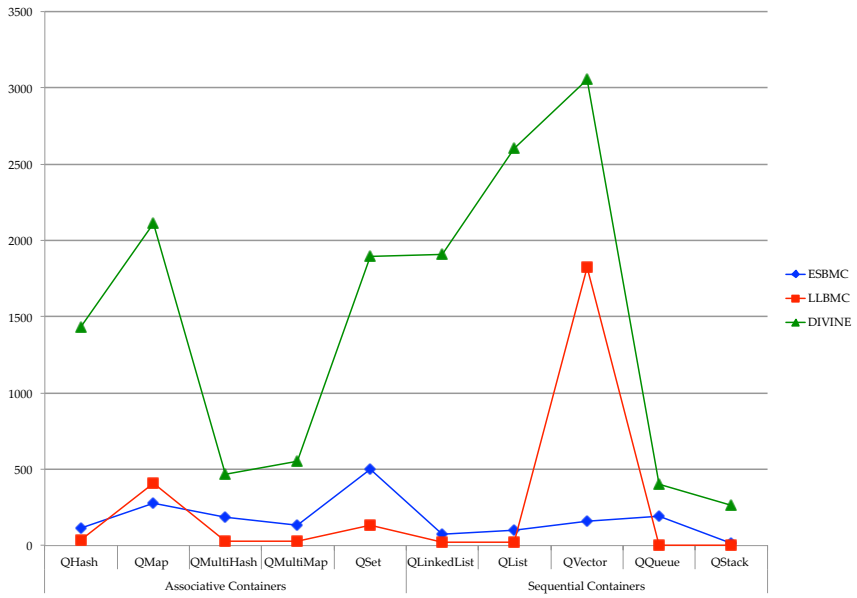


Figura 5.2: Comparação entre os tempos de verificação em relação a ESBMC++, LLBMC e DIVINE.

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 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.

Além disso, embora o QList, QMap, QVector e suites QDefina relataram mais resultados do teste FP e FN com XBMC que as outras opções, a taxa de cobertura sobre casos de teste corretamente verificados, para cada um, foi em torno de 80% a 90%, o que demonstra a sua eficácia em matéria de verificação, uma vez que cada caso de teste verifica uma característica diferente de um recipiente diferente.

Também vale a pena mencionar que ESB C ++ foi capaz de detectar 89,3% dos erros em casos de teste (ou seja, 631 em cada 708 casos de teste), que também demonstra a sua eficácia. Da mesma forma, LLBMC e divina apresentada uma cobertura global de 81,4% e 89% (isto é, 576 e 630 para fora de 708 casos de ensaio), respectivamente, o que demonstra também a adequação em relação à combinação de QtOM com outras ferramentas de verificação. Como consequência, a metodologia proposta não se limita a uma determinada ferramenta e pode ser

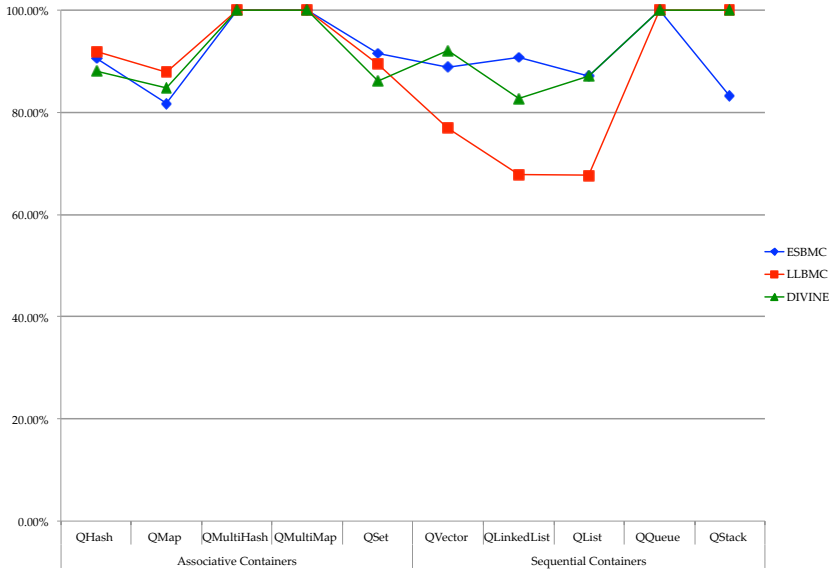


Figura 5.3: Comparação entre a taxa de cobertura em relação a ESBMC++, LLBMC e DIVINE.

adaptado para aplicações específicas, em que algumas abordagens são mais adequados do que outros.

5.3 Resumo

Resumo

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] 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>.

[5] 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>.

[6] 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>.

[7] 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>.

[8] 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.

[9] 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.

[10] 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>.

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

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

[13] 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>.

[14] 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.

[15] 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.

[16] 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>.

[17] 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.

[18] 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>.

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

[20] 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>.

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

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

[23] 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_1$2>.

[24] CORDEIRO, L. C.; FISCHER, B. Verifying multi-threaded software using SMT-based context-bounded model checking. In: *ICSE*. [s.n.], 2011. p. 331–340. Disponível em: <http://doi.acm.org/10.1145/1985793.1985839>.

[25] 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.

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

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

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

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

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

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

[32] 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].

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

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

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

[36] DEITEL, P.; DEITEL, H. *C++ How to Program*. [S.l.]: Prentice Hall, 2013. 1080 p.

[37] 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].

Apêndice A

Publicações

A.1 Referente à Pesquisa

- AINDA FALTA MODIFICAR ESSA PARTE

- **Mikhail Ramalho**, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas Cordeiro e Bernd Fischer. *SMT-Based Bounded Model Checking of C++ Programs*. **20th IEEE International Conference and Workshops on the Engineering of Computer Based Systems**, Phoenix, 2013. p. 147-156.

- **Mikhail Ramalho**, Lucas Cordeiro, André Cavalcante e Vicente Lucena. *Verificação Baseada em Indução Matemática para Programas C/C++*. **III Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Niterói, Rio de Janeiro.

A.2 Contribuições em outras Pesquisas

- AINDA FALTA MODIFICAR ESSA PARTE

- Mauro L. de Freitas, **Mikhail Y. R. Gadelha**, Lucas C. Cordeiro, Waldir S. S. Júnior e Eddie B. L. Filho. *Verificação de Propriedades de Filtros Digitais Implementados com Aritmética de Ponto Fixo*. **XXXI Simpósio Brasileiro de Telecomunicações - SBrT**, 2013.