Verificação de Programas C++ Baseados no *Framework* Multiplataforma Qt

Felipe R. M. Sousa[1], **Lucas C. Cordeiro**[1], **Eddie B. L. Filho**[2]

[1]Centro de Pesquisa e Desenvolvimento de Tecnologia Eletrônica e da Informação
Universidade Federal do Amazonas (UFAM) – Manaus, AM – Brasil

[2]Centro de Ciência, Tecnologia e Inovação do Pólo Industrial de Manaus
Manaus – AM – Brasil

{felipemonteiro,lucascordeiro}@ufam.edu.br, eddie@ctpim.org.br

Abstract. *The development process for embedded systems is getting faster and faster, which generally incurs an increase in the associated complexity. As a consequence, companies in this area usually invest a lot of resources in fast and automatic verification processes, in order to create robust systems and reduce product recall rates. Because of that, the present paper proposes a simplified version of the Qt framework, which is integrated into the Efficient SMT-Based Bounded Model Checking (ESBMC) tool, in such a way that it is able to verify actual applications that use the mentioned framework. The method proposed in this paper presents a success rate of* 91.67% *for the developed test suite.*

Resumo. *O processo de desenvolvimento de sistemas embarcados vem ocorrendo de forma cada vez mais acelerada, o que resulta no aumento da sua complexidade. Como consequência, empresas dessa área normalmente investem muitos recursos em verificação rápida e automática, de modo a gerar sistemas mais robustos e diminuir taxas de retorno de produto. Por essa razão, o presente trabalho propõe um conjunto de bibliotecas simplificadas, similares ao framework Qt, que estão integradas ao verificador de software Efficient SMT-Based Bounded Model Checking (ESBMC), de modo que este seja capaz de analisar aplicações reais que utilizam o framework mencionado. O método proposto apresenta* $91,67\%$ *de acerto para a suite de teste desenvolvida.*

1. Introdução

A dependência do usuário, com relação ao funcionamento correto de sistemas embarcados, está aumentando rapidamente. A grande difusão de dispositivos móveis, aliada à evolução dos componentes de software e hardware que os constituem, são bons exemplos da importância desses sistemas. Eles estão se tornando cada vez mais complexos e requerem processadores com vários núcleos de processamento e memória compartilhada escalonável, com o intuito de atender à crescente demanda por poder computacional. Dessa maneira, a confiabilidade de sistemas embarcados acaba se tornando um assunto chave no processo de desenvolvimento de dispositivos móveis [van der Merwe et al. 2014, Cordeiro et al. 2012a].

Cada vez mais, empresas da área de sistemas embarcados procuram formas mais rápidas e baratas de verificar a confiabilidade de seus sistemas, evitando assim grandes prejuízos [Berard et al. 2010]. Uma das formas mais eficazes e de menor custo é a verificação de modelos [Clarke et al. 1999], porém, apesar das vantagens desse

método, existem muitos sistemas que não podem ser verificados de forma automática, devido à indisponibilidade de verificadores que suportem determinadas linguagens e *frameworks*. Por exemplo, o verificador *Java PathFinder* é capaz de analisar programas Java baseados em Java *byte-code* [National Aeronautics and Space Administration 2007], mas não suporta a verificação de aplicações Java que utilizam o sistema operacional (SO) Android, a menos que se crie um modelo operacional para o SO Android [van der Merwe et al. 2014].

Dessa forma, o presente trabalho visa mapear as principais funcionalidades do *framework* Qt e, a partir do resultado obtido, desenvolver um modelo operacional capaz de verificar as suas propriedades. Os algoritmos desenvolvidos neste artigo estão integrados no verificador de código *Efficient SMT-based Context-Bounded Model Checker* (ESBMC) [Cordeiro and Fischer 2011, Cordeiro et al. 2012a], o qual é reconhecido internacionalmente pela sua robustez e eficácia na verificação de programas ANSI-C [Cordeiro et al. 2012a] e C++ [Ramalho et al. 2013]. De fato, ele ficou em destaque devido à sua atuação nas recentes edições da Competição Internacional em Verificação de Software [Cordeiro et al. 2012b, Morse et al. 2013, Morse et al. 2014].

O respectivo verificador utiliza teorias do módulo da satisfabilidade, do inglês *Satisfiability Modulo Theories* (SMT), aliadas as técnicas de *Bounded Model Checking* (BMC) para verificar, em poucos segundos (dependendo da quantidade de *loops* e *interleavings*) [Cordeiro et al. 2012a, Cordeiro and Fischer 2011], determinadas propriedades de um dado código ANSI-C/C++, tais como *under-* e *overflow* aritmético, segurança de ponteiros, limites de arrays, divisão por zero, vazamento de memória, violações de atomicidade e ordem, *deadlock* e corrida de dados. Além da verificação de tais propriedades, a partir do modelo operacional desenvolvido neste trabalho, o ESBMC será capaz de identificar as estruturas do *framework* Qt (classes, métodos, funções e macros) e verificar as propriedades específicas relacionadas a tais estruturas, denominadas neste artigo de pré- e pós-condições. É importante também ressaltar que não existe outro *model checker* para verificação do *framework* Qt, o que mostra o potencial inovador desta pesquisa.

2. Verificação de Software e Sistemas

Atualmente, a pilha de software em produtos embarcados tem aumentado significativamente, de tal modo que a sua verificação desempenha um papel importante na qualidade final do produto. Por exemplo, as empresas instaladas no Pólo Industrial de Manaus (PIM) têm focado grande parte de seus investimentos, relativos aos recursos gerados com a lei de informática, no desenvolvimento de software para dispositivos móveis. Além disso, diversos módulos de software têm sido utilizados para acelerar o desenvolvimento das aplicações. Dentro desse contexto, o *framework* Qt [Qt Company 2015] representa um bom exemplo de conjunto reutilizável de classes, com o qual um engenheiro de software pode acelerar o desenvolvimento de aplicações gráficas para dispositivos móveis.

É nesse cenário que a verificação formal surge como uma técnica eficiente, no que diz respeito à validação desse tipo de sistema, oferecendo garantia de corretude. Essa técnica tem por objetivo provar, matematicamente, a conformidade de um determinado algoritmo, com relação a uma determinada propriedade, através de métodos formais [Clarke et al. 1999].

A verificação formal é divida em dois tipos de abordagens: a verificação dedutiva e a verificação de modelos. No caso da verificação dedutiva, que funciona para sistemas de espaço infinito, o sistema em análise e as propriedades de corretude são descritos

em conjuntos de fórmulas, de modo que, através da utilização de axiomas e regras de prova, seja possível verificar a corretude destas. Contudo, a tecnologia de provadores de teorema é difícil e não é completamente automática [Clarke et al. 1999]. Por outro lado, no caso da verificação de modelos (*Model Checking*), que tipicamente funciona em sistemas de espaço finito, um sistema é representado por um determinado modelo. Dessa maneira, todas as suas propriedades podem ser checadas, de forma completamente automática [Clarke and Schilingloff 2001]. Para se tratar a segunda abordagem, de forma algorítmica, o sistema sob análise e as suas especificações são representados matematicamente, utilizando-se proposições lógicas (*e.g.*, lógica temporal linear e de ramificação), de tal forma que se possa verificar se uma dada fórmula é satisfeita, de acordo com uma determinada estrutura.

De modo a realizar esse processo, de forma automática, algumas ferramentas de software, conhecidos como verificadores, são utilizados. Eles são desenvolvidos com base em teorias de verificação de modelos, tendo como principal objetivo checar, de forma confiável, as propriedades de um determinado tipo de código. Um exemplo desse tipo de software, o qual foi utilizado nesta pesquisa, é o verificador de modelos ESBMC. Esse verificador, que é baseado em SMT, verifica execuções limitadas de um programa, usando um limite de chaveamento de contexto, através do desdobramento de *loops* e *inlining* de funções, para então gerar um grafo de controle acíclico contendo todas as informações sobre o programa. Esse grafo é convertido em uma fórmula SMT, que é então solucionada com fragmentos das lógicas de primeira ordem, *e.g.*, QF_AUFBV e QF_AUFLIRA [Barrett et al. 2010], [Stump, A. and Deters, M. 2010], dos solucionadores SMT. A partir disso, se existir um erro dentro do código, o ESBMC retornará um contraexemplo, que mostra todo o caminho a ser percorrido para se reproduzir este erro, juntamente com informações sobre classificação e localização [Rocha et al. 2012]. Com esse processo, o ESBMC pode verificar várias propriedades em um determinado código, como *overflow* aritmético, segurança de ponteiros, vazamento de memória, limites do vetor, violações de atomicidade e ordem, *deadlock*, corrida de dados e assertivas especificadas pelo usuário. É importante ressaltar que, em última análise, as propriedades de um código serão checadas através de assertivas.

3. Verificação de Programas C++ Baseados no Framework Qt

Durante o processo de verificação com o ESBMC, o primeiro estágio é o analisador (*parser*). Nesse estágio, o verificador transcreve o código de entrada em uma estrutura de dados, denominada Árvore Sintática Abstrata (ASA), que contém todas as informações necessárias à verificação do algoritmo. Entretanto, para realizar este estágio com sucesso, o ESBMC precisa identificar corretamente todas as estruturas do código a ser verificado.

Como já mencionado, até então o verificador não era capaz de identificar as estruturas de um código que utiliza o *framework Qt*, dado que o ESBMC somente suporta as linguagens ANSI-C e C++. Além disso, por se tratar de um *framework* robusto, o conjunto de bibliotecas padrões do *Qt* contém uma estrutura hierárquica e complexa, com um amplo conjunto de classes, módulos para manipulação de clientes e servidores, métodos e sensores, entre outros, o que tornaria a utilização de tais bibliotecas, durante o processo de verificação, uma abordagem extremamente difícil. Em resumo, o *parser* atual do ESBMC seria incapaz de identificar todas as estruturas existentes, o que inviabilizaria o processo de verificação. Por esta razão, a utilização de um modelo simplificado, contendo apenas as estruturas necessárias para a verificação das propriedades existentes

no *framework* Qt, configura como uma abordagem mais viável para a verificação de programas nesse seguimento. É importante salientar que, para se garantir a verificação das propriedades relacionadas as estruturas do *framework* Qt, assertivas são adicionadas ao modelo operacional, com o objetivo de checar as respectivas propriedades.

O que foi exposto evidencia a necessidade de se desenvolver um modelo operacional, escrito em C++, que proporcione uma representação simplificada do *framework* Qt. A partir da utilização de modelos mais simples, é possível diminuir a complexidade da ASA e, consequentemente, diminuir o custo computacional, de tal forma que o ESBMC possa se basear nesses modelos para, durante a etapa de análise, construir uma ASA que consiga englobar todas as propriedades necessárias à verificação do código, de forma rápida. Além disso, um ponto fundamental, que não é abordado pelo conjunto de bibliotecas padrões do *framework Qt*, é a necessidade de assertivas para a verificação de propriedades relacionadas à execução e também à função de cada método. Tendo isso em mente, à medida que o modelo operacional é desenvolvido, tais assertivas são adicionadas aos seus respectivos métodos e funções, de maneira que, a partir destas, o ESBMC será capaz de verificar propriedades específicas do modelo (*e.g.*, passar um valor maior ou igual a zero para representação de medidas).

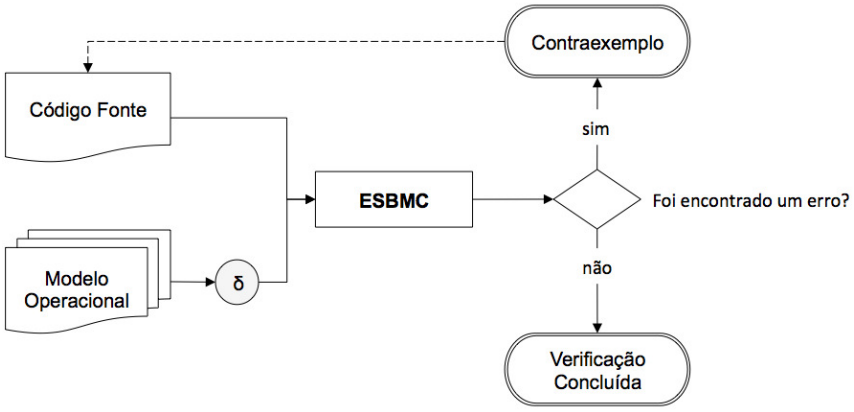

Figura 1. Diagrama do processo de verificação com o modelo operacional.

O modelo operacional foi desenvolvido separadamente do núcleo do ESBMC e é fornecido somente no início do processo de verificação, como pode ser observado na Figura 1. Dessa forma, no momento da verificação, o código fonte a ser checado e o modelo operacional são passados. O modelo é conectado ao ESBMC e também ao respectivo código através de um parâmetro[1] do comando de verificação, representado na Figura 1 por δ. Além dessa informação, o limite de memória [2] e o limite de tempo [3] também são informados, pois estes serão utilizados durante a verificação. Sendo assim, o verificador realizará todo o processo de verificação, retornando como resultado a existência ou não de alguma violação. Além disso, no caso da detecção de um erro, o ESBMC também retornará um contraexemplo, indicando o caminho que deve ser percorrido, durante a execução do programa, para se chegar até ele [Rocha et al. 2012].

[1] –I /libraries/Qt/
[2] −−memlimit 22000000
[3] −−timeout 900

4. Discussão dos Resultados

Na Figura 2, é possível observar imagens referentes à aplicação denominada *Animated Tiles* [Qt Project Hosting 2012a], que é um dos exemplos de código disponíveis na documentação do Qt [Qt Project Hosting 2012a] e foi desenvolvido pelo Instituto Nokia de Tecnologia [Nokia Corporation 2012]. Nessa aplicação, existem alguns ícones no canto inferior direito, que possibilitam ao usuário escolher a disposição das imagens presentes no centro da aplicação. Esse código contém 215 linhas e exemplifica a utilização de várias bibliotecas, contidas no módulo do *framework* Qt denominado *QtGui*, que estão relacionadas ao desenvolvimento de animações e interfaces gráficas.

Figura 2. Imagens de diferentes tipos de disposições providas pela aplicação *Animated Tiles*.

A Figura 3 mostra um trecho do código da aplicação *Animated Tiles*, no qual um objeto da classe $QTimer$ é definido e este, em seguida, chama o método $start()$, que inicia ou reinicia a contagem de um determinado intervalo de tempo (passado como parâmetro, em milissegundos) [Qt Project Hosting 2012b]. Com base nas especificações da documentação do *framework*, o modelo operacional (simplificado) da respectiva classe foi construído, contendo o corpo da mesma e a assinatura dos seus respectivos métodos. Após a definição da estrutura, é necessário que cada método, contendo propriedades a serem verificadas, seja modelado. A utilização das assertivas, na composição desta modelagem, garante a verificação das propriedades relacionadas aos respectivos métodos. Existem várias propriedades que devem ser checadas, como acesso de memória inválido, definição de tempo e tamanho com números negativos, acesso aos arquivos inexistentes e ponteiros nulos. Entre as propriedades que devem ser checadas, há algumas conhecidas como pré-condições, que determinam as condições mínimas para que uma determinada funcionalidade seja utilizada corretamente.

```
1 QTimer timer;
2 timer.start(125);
3 timer.setSingleShot(true);
4 trans = rootState->addTransition(&timer,
5   SIGNAL(timeout()),
6   ellipseState);
7 trans->addAnimation(group);
```

Figura 3. Trecho de código retirado da aplicação *Animated Tiles*.

No trecho de código em questão, por exemplo, o método $start()$ possui uma pré-condição: o valor que é passado como parâmetro deve ser, obrigatoriamente, maior ou

igual a zero, dado que se trata de uma especificação de tempo. Desse modo, uma assertiva é adicionada dentro do método, a qual verifica se o valor segue a especificação e, caso haja alguma violação, o ESBMC indicará que o código contém um erro e, então, enviará uma mensagem especificando as condições para que este ocorra.

É possível observar, na Figura 4, a checagem da pré-condição (linha 6) e o modelo operacional criado para a classe $QTimer$, no qual existe uma implementação do respectivo método (linhas de 5 a 7). Tendo isso em mente, quando o método $start()$ for chamado, o ESBMC entenderá sua implementação como descrita no modelo operacional, *i.e.*, a assertiva checando o parâmetro será chamada. Para o código da Figura 3, o parâmetro é válido e a assertiva é verdadeira ($125 >= 0 \implies true$); entretanto, caso um parâmetro inválido seja passado ao método $start()$, como o valor -125, a assertiva será falsa ($-125 >= 0 \implies false$). Nesse caso, o ESBMC retornará um contraexemplo indicando todos os passos necessários para a reprodução da violação, além do tipo de erro especificado na assertiva (no presente caso, *Time specified invalid*).

```
1  class Qtimer {
2  public:
3    QTimer(){}
4    QTimer ( QMainWindow* window ){}
5    void start ( int i ) {
6      __ESBMC_assert(i >= 0 ,''Time specified invalid.'')
7    }
8    void setSingleShot( bool b ){}
9    void start(){}
10   void stop(){}
11   void setInterval( int msec ){}
12 };
```

Figura 4. Modelo operacional da classe *QTimer*.

Além disso, existem métodos que, do ponto de vista de verificação, não apresentam nenhuma propriedade a ser checada, como métodos cuja única funcionalidade é imprimir um determinado valor. Desse modo, dado que a finalidade do ESBMC é verificar o software e não o hardware, o processo de verificação de corretude do valor impresso não foi abordado neste trabalho. Tais métodos apresentam uma estrutura (assinatura), pois é necessário que o verificador os reconheça durante o processo de análise, de modo a criar uma ASA confiável, porém, não apresentam nenhuma modelagem (corpo), dado que não existem propriedades a serem verificadas.

Por outro lado, existem métodos que, além de apresentarem propriedades que precisam ser analisadas como pré-condições, também apresentam propriedades que devem ser analisadas como pós-condições. Um exemplo disso está presente no código da Figura 5, onde elementos são inseridos no início de uma determinada lista ($mylist$) e, posteriormente, o primeiro elemento é retirado. Em seguida, o início da lista é checado, através de uma assertiva. A partir dos modelos operacionais dos métodos $push\_front()$ e $pop\_front()$, na Figura 6, é possível verificar que, se apenas as pré-condições forem analisadas, não existirá nenhum indício, no modelo, que valores foram adicionados ou retirados da respectiva lista e, consequentemente, a ASA não teria nenhuma representação de tais valores. Desse modo, as assertivas, encontradas nas linhas 9 e 11 da Figura 5, não existiriam, implicando uma inconsistência no processo de verificação.

```
#include <iostream>
#include <cassert>
#include <QList>
using namespace std;
int main () {
  QList<int> mylist;
  mylist.push_front(200);
  mylist.push_front(300);
  assert(mylist.front() == 300);
  mylist.pop_front();
  assert(mylist.front() == 200);
  return 0;
}
```

Figura 5. Exemplo de código utilizando a biblioteca *QList*.

Nesse contexto, é necessário implementar o comportamento do respectivo método, de maneira que a verificação das propriedades relacionadas à manipulação ou ao armazenamento de valores, no decorrer de um programa, seja feita de forma consistente. É importante ressaltar que a execução de determinados métodos e funções implicará a modificação de estruturas do programa (*e.g.*, *containers*), portanto, é fundamental que se mantenha, no modelo, uma representação consistente de tais estruturas, de modo que a verificação das pós-condições seja feita corretamente (resultados da execução de métodos e funções). Nesse caso em particular, como pode ser observado na implementação do modelo do método $pop\_front()$, na Figura 6, além de checar, através de uma assertiva, se a lista em questão não está vazia (análise da pré-condição), a retirada do elemento da lista (simulação do comportamento do método) também é implementada, o que enriquece o poder de representação do modelo.

```
void push_front( const value_type& x ){
  if(this->_size != 0) {
    for(int i = this->_size -1; i > -1; i--)
      this->_list[i+1] = this->_list[i];
  }
  this->_list[0] = x;
  this->_size++;
}
void pop_front() {
  __ESBMC_assert(!empty(),``The list can not be empty.'');
  this->_size--;
  for(int i = 0; i < this->_size; i++){
    this->_list[i] = this->_list[i+1];
  }
  this->_list[this->_size + 1] = value_type();
}
```

Figura 6. Exemplo de modelos de métodos, cujas funcionalidades são simuladas no processo de verificação.

Com base no que foi exposto até agora, percebe-se que a implementação do mo-

delo operacional deve ser realizada com bastante cuidado e seguindo-se a descrição fornecida pela documentação oficial. Além disso, para que haja confiança nas bibliotecas disponibilizadas, estas são manualmente testadas e comparadas as originais (através de testes unitários), de modo a se garantir o mesmo comportamento.

A partir de análise de alguns *benchmarks* fornecidos pelo Instituto Nokia de Tecnologia, observou-se o uso constante de bibliotecas pertencentes aos módulos $QtGui$ e $QtCore$, que contém as definições gráficas e as classes base para outros módulos, respectivamente. A partir disso, uma suíte de teste foi construída, denominada *esbmc-Qt*, que verifica, de forma automática, todo o código a ela adicionado, além de simular as bibliotecas dos módulos citados anteriormente.

À medida que os modelos operacionais foram desenvolvidos, *benchmarks* também foram adicionados à respectiva suíte, com o objetivo de validar as suas implementações. Todos os experimentos foram realizados em um computador *Intel Core* $i7$-2600, com $3,40$ GHz de clock e 24 GB de RAM, executando o sistema operacional Ubuntu (64 bits) e o ESBMC[4] 1.20. O limite de tempo e memória, para cada caso de teste, foram fixados em 900 segundos e 24 GB (22 GB de RAM e 2 GB de memória virtual), respectivamente[5]. Os tempos indicados foram medidos através do comando $time$. Atualmente, a suíte de teste contém 52 *benchmarks* (1671 linhas de código), o que leva 414 segundos para ser verificado: $91,67\%$ são checados corretamente e $8,33\%$ apresentam um resultado "falso negativo", o que ocorre quando um caso não contém erro e o verificador indica o contrário. Tais erros estão relacionados com a representação interna de ponteiros do verificador. O ESBMC, uma vez que é baseado em BMC, reduz o *trace* limitado de um programa a lógica de primeira ordem, o que implica em limitar a profunidade da análise e eliminar ponteiros no verificador de modelos. Contudo, durante este processo, podem surgir algumas inconsistências que provocam os erros identificados na verificação. Além disso, modelos operacionais referentes a 128 bibliotecas do *framework* Qt foram desenvolvidos, totalizando 3483 linhas de código.

5. Trabalhos Relacionados

No segmento dos verificadores de modelos, não existe outro verificador conhecido para o *framework* Qt. Por essa razão, este artigo não apresenta uma comparação com outro verificador. No entanto, a verificação de software através da técnica BMC é utilizada por diversas ferramentas (*e.g.*, CBMC [Kroening and Tautschnig 2014]) e está se tornando cada vez mais popular, principalmente devido ao surgimento de solucionadores SMT sofisticados, que são desenvolvidos com base em solucionadores de satisfabilidade (SAT) eficientes [de Moura and Bjørner 2008].

Nesse segmento, é possível destacar a ferramenta *Low-Level Bounded Model Checker* (LLBMC) descrita por Mers *et al.*, que também aplica a técnica BMC à verificação de programas ANSI-C/C++ [Falke et al. 2013]. Essa ferramenta utiliza o compilador LLVM para converter programas ANSI-C/C++ na representação intermediária LLVM, que por sua vez perde informações sobre a estrutura dos respectivos programas em C++ (*i.e.*, as relações entre classes). De forma similar ao ESBMC, Merz *et al.* também aplica solucionadores SMT para checar as condições de verificação, do inglês *verification conditions* (VCs), que são geradas a partir de programas C++. No entanto, diferentemente da abordagem proposta neste trabalho, o LLBMC não suporta tra-

[4]http://www.esbmc.org

[5]−−unwind 10 −−no−unwinding−assertions −I /libraries/Qt/ −−memlimit 22000000 −−timeout 900

tamento de exceções, o que torna difícil a verificação de programas reais escritos em C++ (*e.g.*, programas que dependem do conjunto de bibliotecas *Standard Template Libraries* – STL) [Ramalho et al. 2013].

Blanc *et al.* descrevem a verificação de programas C++ que utilizam o conjunto de bibliotecas *Standard Template Libraries* (STL), via abstração de predicado [Blanc et al. 2007]. A técnica proposta visa a utilização de tipos de dados abstratos para a verificação do uso das bibliotecas STL, ao invés da aplicação do próprio conjunto STL. Blanc *et al.* mostra que é suficiente verificar a corretude de programas usando um modelo operacional, uma vez que a prova das pré-condições, sobre uma determinada operação no modelo, implicará a garantia das mesmas pré-condições especificadas pela linguagem, para a respectiva operação. Essa abordagem é eficiente para encontrar erros triviais em programas C++, porém, apresenta uma deficiência na pesquisa mais aprofundada de erros em operações enganosas (*i.e.*, quando se trata da modelagem interna dos métodos). Neste trabalho, isso é contornado através da simulação do comportamento de determinados métodos/funções (ver seção 4).

6. Conclusão

O objetivo deste trabalho foi desenvolver um modelo operacional capaz de representar, de forma simplificada, o *framework* multiplataforma Qt e suas propriedades e então integrá-lo ao processo de verificação de código C++/Qt do ESBMC. É importante ressaltar que esta ferramenta representa o estado da arte em verificação de software e que não existe outro verificador de modelos para o *framework* Qt.

Dois módulos do *framework* Qt foram abordados: *QtGui* e *QtCore*. Além disso, uma suíte de testes automatizada foi desenvolvida, com o objetivo de validar as implementações realizadas, as quais contém código englobando diversas bibliotecas dos respectivos módulos. Vale ressaltar que a abordagem utilizada durante o processo de desenvolvimento do modelo operacional foi apresentada, juntamente com os resultados obtidos através das verificações realizadas na suíte de teste automatizada, totalizando uma taxa de $91,67\%$ de verificações corretas.

Por fim, alguns trabalhados futuros a serem desenvolvidos, relativos a esta pesquisa, são o aumento da variedade de bibliotecas representadas no modelo operacional, o que possibilitaria uma maior cobertura do *framework*, a identificação de outras propriedades a serem checadas, o que ampliaria o número de casos de teste, e a inclusão de benchmarks e aplicações reais.

Referências

Barrett, C., Stump, A., and Tinelli, C. (2010). The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org. Acesso em: Agosto, 2012.

Berard, B., Bidoit, M., and Finkel, A. (2010). *Systems and Software Verification: Model-Checking Techniques and Tool*. Springer Publishing Company, 1st edition.

Blanc, N., Groce, A., and Kroening, D. (2007). Verifying C++ with STL containers via predicate abstraction. In *ASE*, pages 521–524.

Clarke, E. M., Grunberg, O., and Peled, D. (1999). *Model Checking*. Springer Publishing Company, 1st edition.

Clarke, E. M. and Schilingloff, H. (2001). *Handbook of Automated Reasoning*, chapter Model Checking, pages 1635–1790. The MIT Press, 1st edition.

Cordeiro, L., Fischer, B., and Marques-Silva, J. (2012a). SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974.

Cordeiro, L. C. and Fischer, B. (2011). Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE*, pages 331–340.

Cordeiro, L. C., Morse, J., Nicole, D., and Fischer, B. (2012b). Context-bounded model checking with ESBMC 1.17 - (competition contribution). In *TACAS*, volume 7214 of *LNCS*, pages 534–537.

de Moura, L. M. and Bjørner, N. (2008). Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340.

Falke, S., Merz, F., and Sinz, C. (2013). The bounded model checker LLBMC. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 706–709.

Kroening, D. and Tautschnig, M. (2014). CBMC - C bounded model checker - (competition contribution). In *TACAS*, volume 8413 of *LNCS*, pages 389–391.

Morse, J., Cordeiro, L. C., Nicole, D., and Fischer, B. (2013). Handling unbounded loops with ESBMC 1.20 - (competition contribution). In *TACAS*, volume 7795 of *LNCS*, pages 619–622.

Morse, J., Ramalho, M., Cordeiro, L. C., Nicole, D., and Fischer, B. (2014). ESBMC 1.22 - (competition contribution). In *TACAS*, volume 8413 of *LNCS*, pages 405–407.

National Aeronautics and Space Administration (2007). Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/. Acesso em: Abril, 2015.

Nokia Corporation (2012). NOKIA Global. http://www.nokia.com/global.html. Acesso em: Setembro, 2012.

Qt Company (2015). The Framework. http://www.qt.io/qt-framework/. Acesso em: Abril, 2015.

Qt Project Hosting (2012a). Animated Tiles Example. http://doc.Qt.digia.com/Qt/animation-animatedtiles.html. Acesso em: Setembro, 2012.

Qt Project Hosting (2012b). QTimer Class Reference. http://Qt- project.org/doc/Qt-4.7/Qtimer.html. Acesso em: Setembro, 2012.

Ramalho, M., Freitas, M., Sousa, F., Marques, H., Cordeiro, L. C., and Fischer, B. (2013). SMT-Based Bounded Model Checking of C++ Programs. In *ECBS*, pages 147–156.

Rocha, H., Barreto, R. S., Cordeiro, L. C., and Neto, A. D. (2012). Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *IFM*, volume 7321 of *LNCS*, pages 128–142.

Stump, A. and Deters, M. (2010). Satisfiability Modulo Theories Competition (SMT-COMP). http://smtcomp.sourceforge.net/2012/index.shtml. Acesso em: Agosto, 2012.

van der Merwe, H., van der Merwe, B., and Visser, W. (2014). Execution and property specifications for jpf-android. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5.