Verificação de Programas Multi-Thread Baseados no Framework Multiplataformas Qt

Adriana Silva de Souza[1], Lucas C. Cordeiro[2], Francisco A. P. Januario[1]
[1]Fundação Centro de Análise, Pesquisas e Inovação Tecnológica (FUCAPI) – Manaus – AM – Brasil.
[2]Centro de Pesquisa e Desenvolvimento de Tecnologia Eletrônica e da Informação
Universidade Federal do Amazonas (UFAM) – Manaus, AM – Brasil

adrianasilvadesouza0@gmail.com,lucascordeiro@ufam.edu.br,
francisco.januario@fucapi.br

Resumo. *Com o avanço da tecnologia, progressos tem acontecido no que diz respeito ao desenvolvimento de hardware e software. O resultado de tais avanços é decorrente da evolução das aplicações, que atualmente requerem mais capacidade de processamento e armazenamento de dados. A partir dessa realidade, a indústria investe fortemente em processos de verificação automáticos e rápidos, com o intuito de obter a diminuição das taxas de erros presentes nos sistemas produzidos. O presente trabalho propõe um conjunto de bibliotecas simplificadas, similares ao framework Qt, que integradas ao verificador de software Efficient SMT-Based Bounded Model Checking (ESBMC), de forma que através do modelo seja possível analisar aplicações reais que utilizam o framework Qt para implementar programação concorrente.*

1. Introdução Geral

Falhas são cada vez mais inaceitáveis em sistemas embarcados, embora sejam inevitáveis, podem ser mitigadas com o uso de técnicas adequadas. A confiabilidade dos sistemas de software está tornando-se uma questão chave no processo de desenvolvimento de sistemas. Na busca por atender a grande demanda, tais sistemas estão tornando-se cada vez mais robustos e complexos, sendo exigido cada vez maior poder computacional, com processadores com memórias escalonáveis é de vários núcleos. Dessa maneira, a confiabilidade de sistemas embarcados torna-se um assunto chave no processo de desenvolvimento de sistemas embarcados [van der Merwe et al. 2014, Cordeiro et al. 2012a]. Portanto, a construção de sistemas de software torna-se um desafio cada vez mais complexo para os engenheiros de software, principalmente quando trata-se de sistemas de software pertencentes a sistemas críticos como sistemas hospitalares, aéreos, automobilísticos.

Sistemas críticos necessitam ser fortemente verificados para a identificação de erros, que podem resultar em falhas durante a execução do software. Assim sendo, para o desenvolvimento de software de alta qualidade, as técnicas de verificação de software são indispensáveis. 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 que utiliza a técnicas de verificação de modelos [Clarke et al. 1999]. Devido a possibilidade de automatizar os testes, a verificação de modelos tem se

tornado uma abordagem muito interessante, cuja simplificação se baseia na verificação de dado modelo á partir de sua especificação.

Apesar das vantagens desse método, nem todos os sistemas computacionais podem ser verificados de forma automática, uma vez que os verificadores devem suportar as linguagens e propriedade do sistema, o que incluem as bibliotecas vinculadas do sistema a ser verificado. Um bom exemplo é o verificador *Java PathFinder*, que é capaz de verificar executáveis de programa Java baseados em Java *byte-code* [National Aeronautics and Space Administration 2007], no entanto não suporta a verificação, de programas escritos em *Java* que utilizem o sistema operacional *android*, a menos que se crie um modelo operacional(MO) para o SO *Android* [van der Merwe et al. 2014], através da representação abstrata das bibliotecas associadas, o mesmo deve aproximar-se das bibliotecas originais para fornecer as mesmas entradas e saídas que a aplicação principal. [Monteiro et al. 2015].

O presente trabalho propõe um modelo operacional para estruturas de dados do *framework* Qt, que implementem programação concorrente. Com isso, fornecer um modelo simplificado *multi-threads*, capaz de verificar as principais propriedades relacionadas aos módulos *Qthreads* e *Qmutex* presentes em tal framework, possibilitando assim, a verificação de concorrência e sincronização dos dados. O modelo simplificado *multi-threads* está integrado ao ESBMC, com o intuito de verificar propriedades especificas de programas concorrentes Qt/C++. Tais propriedades foram verificadas através da inclusão de pré e pós-condições que foram adicionadas no código. Uma aplicação do mundo real foi incorporada ao conjunto de *benchmarks*

Contribuições. O trabalho é estendido de um trabalho anterior [*ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Application*]. Aqui, os aspectos de implementação e o seu uso são discutidos, especialmente, o fato de nessa ocasião o *ESBMC^{QtOM}* incluir novos recursos essenciais dos módulos *Qthread* e *Qmutex*: Incluindo o *benchmark Semaphoro e o benchmark Mandelbrot* que não fazia parte do conjunto de *benchmarks* anteriores. Dessa forma a principal contribuição desse trabalho é atender as principais funcionalidades dos módulos *Qthreads* e *QMutex* presentes no *framework* Qt. Partindo dos resultados obtidos, foi desenvolvido um módulo para o *ESBMC^{QtOM}* capaz de verificar as propriedades presentes em tais módulos do *framework Qt*. Os códigos desenvolvidos nesse trabalho foram integrados ao verificador *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 a 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 ESBMC é um verificador de execução simbólica limitado ao contexto que permite a verificação de códigos simples e *multi-threads* escritos em C++. O ESBMC pode verificar programas que fazem uso de operações com matrizes, ponteiros, estruturas, alocação de memória e aritmética de ponto flutuante. O ESBMC pode detectar, *overflows* aritméticos, segurança de ponteiro, vazamentos de memória, violações de limites de *arrays*, atomicidade e violações de ordem, impasses locais e globais, corridas de dados e afirmações especificadas pelo usuário [Cordeiro and Fischer 2011, Cordeiro et al. 2012a]. O respectivo verificador utiliza a técnica *Bounded Model Checking*(BMC) para gerar as condições para a verificação de determinadas condições,

para um determinado programa, em seguida, converte as condições de verificações, utilizando diferentes teorias de *background*, é transmite para os solucionadores *Satisfiability Modulo Theories*(SMT). O BMC tem como ideia básica que dada uma profundidade, seja verificada a negação de determinadas propriedades. Com o intuito de superar o aumento da capacidade dos software, solucionadores SMT são muito usados como back-ends com o intuito de resolver as condições de verificação(CV) geradas.

O ESBMC suporta nativamente os solucionadores *Z3* e *Boolector*, com isso suporta a análise de código *multi-threads* escritos em ANSI-C que utilizam sincronizações primitivas da biblioteca qthreads POSIX.

A partir do trabalho realizado além das propriedades que o *ESBMC*^{QtOM} já verifica, a partir do modelo simplificado *multi-threads* desenvolvidos nesse trabalho, o *ESBMC*^{QtOM} será capaz de identificar classes, métodos, funções e macros, diretivas de pré-processamento, dos módulos *Qthread* e *Qmutex*, sendo assim possível verificar propriedades especificas presentes nessas estruturas. Ressaltando que não e existem modelos operacionais capazes de verificar tais módulos, ressaltando assim a importância da pesquisa realizada no trabalho.

2. Verificação Formal de Software

A quantidade de sistemas de software embarcados, vem aumentando substancialmente, cada vez mais complexos e robustos, esses sistemas tem sido um desafio para os engenheiros e projetistas de sistemas de software embarcados, que buscam sempre a segurança e confiabilidade no produto final. Dessa forma, a verificação de software tem desempenhado um papel importante no produto que é desenvolvido. As empresas têm investido grande parte de seus recursos e esforço nos processos de verificação. Diversos *framework*s têm sido usados com o intuito de acelerar o desenvolvimento desses produtos e partindo desse ponto o *framework* Qt representa um excelente conjunto de classes reutilizáveis, tornando possível desenvolver bibliotecas e aplicativos, sendo possível compilá-los em diversas plataformas sem a necessidade de alterar o código fonte, podendo ser utilizado em ambientes desktop KDE e dispositivo móveis da Nokia, entre outros.

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

É nesse cenário que a verificação formal surge como uma técnica eficiente, no que diz respeito a validação de sistema embarcados críticos, oferecendo garantia de corretude [Monteiro et al. 2015]. 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 deriva-se em verificação dedutiva e verificação de modelos. A verificação dedutiva, dá-se através de um conjunto de fórmulas que descrevem as propriedades dos sistemas, utilizando axiomas e regras de provas, funciona bem com espaços de estados finitos, muito embora, seja difícil realizar a prova de teoremas é não seja completamente automático. De forma oposta, a verificação de modelos checa as propriedades dos modelos, executando exaustivas explorações de

todos os comportamentos possíveis, funcionando apenas em modelos de estados finitos, sendo usado fortemente para a verificação de propriedades de corretude de sistemas de espaços de estados finitos de forma completamente automática.

Na verificação de modelos, o sistema sob análise e as suas especificações são representados matematicamente, utilizando-se proposições lógicas (logica temporal linear e de ramificação), de tal forma que se possa verificar se uma dada formula é satisfeita, de acordo com uma determinada estrutura [Monteiro et al. 2015].

Um grande desafio da verificação formal encontra-se nas explosões de estados, que pode ocorrer caso o sistema contenham muitos componentes interagindo paralelamente, ou mesmo, caso a estrutura de dados assuma valores variados. Nesses casos os estados globais crescem de forma exponencial com o número de processos.

Para realizar a verificação de forma automática verificadores de *software* são utilizados. Os verificadores são em grande parte desenvolvidos tendo como base principal as teorias de verificações de modelos, tendo como objetivo principal realizar de forma confiável, a checagem das principais propriedades dos sistemas. Um software que tem destacando-se e o verificador de modelos ESBMC.

O verificador de modelos ESBMC, é baseado no *front-end Bounded Model Checker for C and C++ Programs*(CBMC), suportando diferentes teorias e solucionadores SMT. Explora informações de alto nível para simplificar e reduzir o tamanho das formulas geradas.

3. Verificação de Programas C++ Baseados no Modelo Simplificado Multi-Threads Qt

Por se tratar de um framework robusto, o conjunto de bibliotecas padrões do *framework* Qt contém uma estrutura hierárquica grande 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 [Monteiro et al. 2015]. Além disso o parse do ESBMC não e capaz de analisar todas as estruturas presentes no *framework* Qt, o que tornaria inviável a verificação. Dada a complexidade de tal *framework* a utilização de um modelo simplificado, contendo apenas as estruturas que serão utilizadas durante a verificação das propriedades foi utilizado, o que a torna a abordagem muito viável no processo de verificação de estruturas que utilizem o *framework* Qt.

Logo com o objetivo de fazer o ESBMC reconhecer estruturas do *framework* Qt que implementem o paralelismo e a concorrência de dados, foi necessário a representação simplificada dos módulos *Qthread* e *Qmutex*, sendo assim, ouve a necessidade do desenvolvimento de um modelo operacional simplificado que atenda a tais estruturas do *framework*, denominado de modelo simplificado *multi-threads*, desenvolvido em C++.

Com isso tornou-se possível diminuir a complexidade da Árvore de Sintaxe Abstrata, diminuindo assim o custo computacional no processo de verificação.

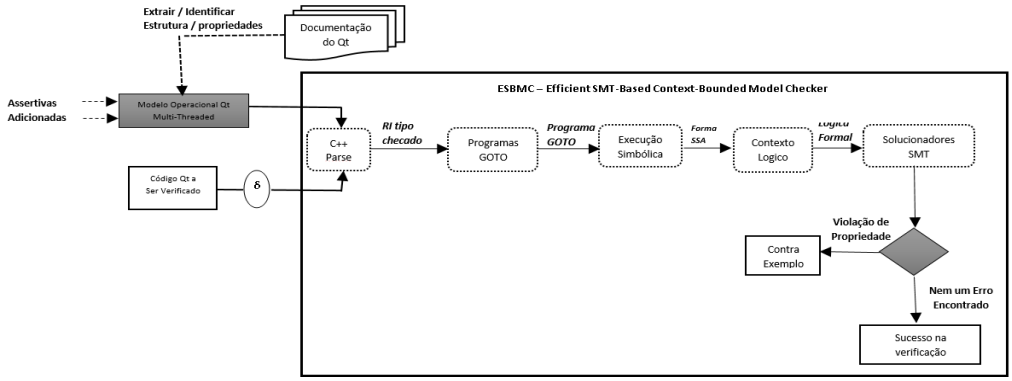

Figura 1 – Fluxograma do Funcionamento do Modelo Simplificado *Multi-Threads* Qt com o ESBMC

A Figura 1, mostra a integração do modelo simplificado *multi-treads* Qt com o ESBMC, na Figura 1 também é possível observar como ocorre o processo de verificação proposto para o *framework* Qt.

A caixa em cinza representa o modelo simplificado *multi-threads* Qt, as pontilhadas representam o ESBMC e as brancas representam as entradas e saídas. Na primeira etapa da verificação, o modelo operacional *multi-threads* e o programa a ser verificado são conectados aos ESBMC, através do parâmetro δ.

No processo de verificação com o ESBMC primeiramente o programa a ser verificado passa pelo analisador Parse, os programa são convertidos em árvore de representação intermediaria (do inglês, *Intermediate Representation* – IRep), em seguida, a árvore de representação intermediaria é convertida em um programa goto (Por exemplo: Expressões do tipo if e while são substituídas por expressões GOTO), que em seguida, são executadas de maneira simbólica pelo GOTO-*symex*, tendo como resultado, a criação de uma atribuição estática(do inglês, *Single Static Assignment* - SSA) única, onde são considerados as classes associadas, incluindo atributos, assinaturas de métodos e afirmações. Tendo como base a ASS o ESBMC realiza uma execução simbólica do programa que está sendo verificado, é gera equações SMT para restrições (atribuições e premissas de variáveis) e propriedades (condições de segurança), onde são posteriormente verificas por solucionadores SMT.

Então caso alguma violação de propriedade seja detectada durante o processo de verificação, o ESBMC retorna um contra exemplo relatando a linha em que o erro foi encontrado, as propriedades que foram violadas e as etapas que foram executas durante a execução, caso contrário o ESBMC retorna que a verificação foi bem sucedida.

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

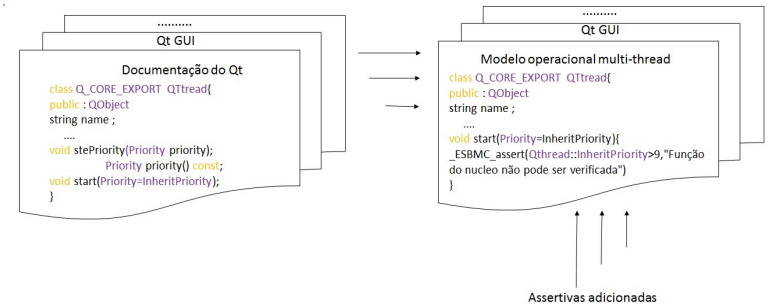

Figura 2 – Processo de Desenvolvimento Modelo operacional *multi-threads* Qt

A Figura 2, representa o processo de desenvolvimento do modelo operacional *multi-threads*, como já mencionado ao longo do desenvolvimento do modelo operacional *multi-threads*, assertivas foram adicionadas aos métodos e funções do *framework* Qt, para que propriedades relacionadas a execução do código possam ser verificadas. Com intuito de detecta violações relacionadas ao uso do *framework* Qt, tais assertivas são integradas nos respectivos métodos e funções, é a partir delas o ESBMC é capaz de verificar tais métodos, funções, as pré e pos-condições, a partir do modelo operacional desenvolvido.

É importante ressaltar que a metodologia proposta baseia-se no fato de que o modelo operacional *multi-threads* representa de maneira correta as bibliotecas aqui representadas do framework Qt. Dessa forma, todos os módulos desenvolvidos no presente trabalho foram testados manualmente e comparados com os originais, com o intuito de garantir que possuam o mesmo comportamento.

4. Discussão dos Resultados

Neste trabalho, foram explorados os módulos *Qthread* e *Qmutex* do *framework* Qt, que estão interligados para assegurar o paralelismo e a concorrência entre os processos. O *framework* Qt fornece mecanismos de baixo e de auto nível para a sincronização de *threads*, possibilitando assim o processo de exclusão mutua. As bibliotecas *QThreads* e *Qmutex* são algumas das bibliotecas disponíveis no referido *framework* que possibilitam a manipulação dos *threads* e a implementação de programas *multi-Threads*.

No Qt cada *thread* tem sua própria pilha de execução, o que significa que cada *thread* tem seu próprio histórico de chamadas e variáveis locais. No diagrama da Figura 3 é possível ver como os *threads* estão localizados na memória.

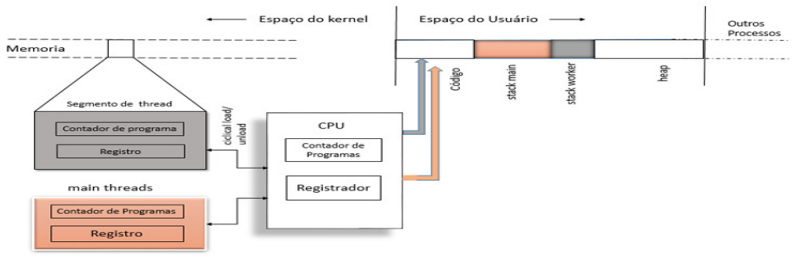

Figura 3. Funcionamento do *thread* no *framework* Qt

Os registros dos threads inativos e os contadores de programas são tipicamente mantidos no espaço do *kernel*. Existe uma cópia compartilhada do código em uma pilha

separada para cada *thread*. Programas que utilizam *multi-threads* em geral são difíceis de testar em grande parte pela sua execução não determinística, o que aumenta potencialmente os espaços de estados. O verificador ESBMC é capaz de verificar programas *multi-threads* utilizando a biblioteca *pthread* nativa do C++, por essa razão, durante a construção do modelo simplificado *multi-threads* foram criadas bibliotecas abstratas dos respectivos módulos do *framework* Qt utilizando a biblioteca *pthread* na criação e manipulação dos *threads*. No ESBMC, o programa analisado é modelado como uma *tupla* onde, são considerados um conjunto finito de estados, assumindo todas as variáveis ao longo de um domínio finito. O respectivo verificador considera programas *multi-threads* de programas assíncronos, presumindo que a comunicação entre os *threads* ocorra através de variáveis compartilhadas. Garantindo que em qualquer momento um único *thread* estará em execução até que ocorra uma mudança de contexto e outro *thread* assuma a execução.

Na figura 3, é possível a observação da imagem da aplicação *Mandelbrot[Mandelbrot Example* (2016a)], *que faz parte dos exemplos disponíveis na documentação do Qt[Mandelbrot Example* (2016a)]. A referida aplicação suporta o zoom e a rolagem de imagens, usando o mouse ou teclado. Essa aplicação contem 510 linhas de código, é também faz uso de diversas bibliotecas presente no modulo do *framework* Qt denominado QtCore, que está relacionado com o desenvolvimento de *multi-threads*.

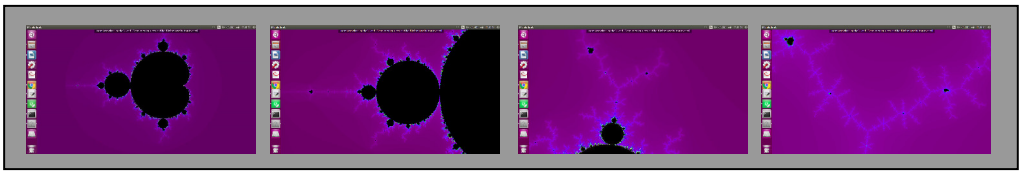

Figura 3. **Imagem onde foi aplicado o zoom pela aplicação *Mandelbrot***

Logo como já mencionado anteriormente seguindo rigorosamente as especificações do framework, o modelo operacional simplificado foi desenvolvido, contendo a assinatura e o corpo de seus métodos. Sendo assim modelado, cada método contendo propriedades especificas que serão verificadas. Dentre as propriedades a serem verificadas estão as pré-condições, a qual estabelece condições mínimas, para a utilização correta utilização de uma determinada funcionalidade.

Na figura 4, e possível observar um pequeno trecho do código da aplicação *Mandelbrot* onde um método chamado *start*() da classe *Qthread*, é chamado pelo método *RenderThread* presente na aplicação, o método *start*() é o responsável de estartar as threads, e nele podem ser definidas as propriedades que cada thread assumira.

```
1  void RenderThread::render (double centerX, double centerY, double scaleFactor,  QSize resultSize) {
2        QMutexLocker locker(&mutex);
3              this->centerX = centerX;  this->centerY = centerY;     this->scaleFactor = scaleFactor;
4              this->resultSize = resultSize;
5     if (!isRunning()) {
6              start(LowPriority);
7     } else {
8              restart = true;
9              condition.wakeOne();
10    }
11 }
```

Figura 4. **Trecho retirado do codigo da aplicação *Mandelbrot***

No trecho de código da Figura 4, o método *start*() possui uma pré-condição: O método start possui oito restrições de prioridade de execução, que podem ser atribuídas a cada thread no momento da execução. Tais propriedades são do tipo *enum*, podendo assim assumir valores inteiros, no modelo operacional *multi-threads* para tais prioridades foram atribuídos valores inteiros de 1 a 8 para identificar cada restrição mais facilmente. Logo obrigatoriamente o valor passado ao método *start*() precisa estar obrigatoriamente entre esses valores de restrição. Partindo daí, para que seja verificado se o valor segue a especificação, assertivas foram inseridas dentro do código, caso violações de propriedades ocorram, o ESBMC indicara que ocorreu um erro, é indicara uma mensagem contendo especificações necessárias para que o erro ocorra.

Na figura 5, observe a checagem da pré-condição (linha 16), em um pequeno trecho do modelo operacional criado para a classe *Qthread*, no qual o respectivo método foi implementado (linhas de 13 a 17). Tendo isso em vista, no momento em que o método *start*() for testado pelo ESBMC, o mesmo entendera sua implementação, a assertiva que está checando o parâmetro será chamada. Considerando o código da Figura 4, a assertiva válida o parâmetro, sendo assim uma verificação bem sucedida. No entanto, caso fosse passado um parâmetro para o método *start*() com o valor 9, a assertiva seria falsa.

```
1   class QThread {
2   private:
3      pthread_t _id;  pthread_attr_t _attr;
4   public:
5   QThread(){}
6   ~QThread(){}
7         enum Priority {
8         IdlePriority=1,LowestPriority=2,LowPriority=3,NormalPriority=4,
9         HighPriority=5, HighestPriority=6, TimeCriticalPriority=7,InheritPriority=8};
10
11  void start(Priority = InheritPriority)
12        {
13           void start ( int i ) {
14           __ESBMC_assert ( i >= 0 , " Priority specified invalid." );
15        }}
```

Figura 5. Trecho do modelo operacional *multi-threads* para a classe *Qthread*

Sendo assim, considerou se na construção do modelo operacional *multi-threads* que os *threads* dos programas comunicam-se por intermédio de variáveis globais e bibliotecas abstratas inseridas no modelo operacional. Para que o ESBMC fosse capaz de entender a concorrência de dados implementadas pelo *framework* Qt, algumas listas de comandos e classe foram incluídas no modelo operacional *multi-thrads*.

As prioridades dos *threads* são criadas através de comunicação assíncrona, que funciona como um sinal onde os dados são transmitidos em um fluxo de execução estável, retornando assim um número inteiro que pode ser utilizado como um identificador para o *thread*, possibilitando assim a sincronização dos dados.

A partir do que foi exposto, é de fundamental importância a implantação correta do comportamento das respectivas classes e métodos expostos, de forma que a verificação das propriedades, ao longo de um programa possam ser verificadas corretamente. Uma importante observação está no fato de que para determinados métodos e funções faz-se necessária a modificação estrutural do programa, de forma que, sejam empregadas representações solidas de tais estruturas de dados.

A construção do modelo operacional *multi-threads* está em andamento, seguindo rigorosamente as descrições da documentação oficial do *framework* Qt. Foram

construídas duas suítes de testes, que foram integradas ao ESBMC, que realiza a verificação de maneira automática dos códigos que estão presentes nas suítes mencionadas. Os testes foram realizados em computador IntelCore i7-2600 com 2.3 GHz de clock e 8 GB de RAM, com sistema operacional Linux Ubuntun versão 14.04 LTS (64 bits).

5. Trabalhos relacionados

No campo de verificação de modelos, não e conhecida a existência de verificadores de *software* capazes de realizar a verificação de tais estruturas do *framework* Qt. Sendo assim, inviabilizado a comparação com outros verificadores. Entretanto, diversas ferramentas realizam a verificação de *software* através da técnica BMC, a mesma tem se popularizado, tendo como motivação principal o crescente surgimento de sofisticados solucionadores SMT.

Nesse campo, a ferramenta *Low-Level Bounded Model Checker* (LLBMC) e um bom exemplo, a ferramenta LLBMC adota a técnica BMC no processo de verificação de programas escritos em ANSI-C/C++. Essa ferramenta utiliza o compilador LLVM para converter programas ANSI-C/C++ na representação intermediaria LLVM, que por sua vez perde informações sobre a estrutura dos respectivos programas em C++ (i.e., as relações entre classes) [Monteiro et al. 2015]. De maneira semelhante o ESBMC também faz uso de solucionadores SMT, para checar as CVs geradas a partir de programas escritos em C++. No entanto, de forma oposta a abordagem que e proposta no presente trabalho, o LLBMC não oferece suporte ao tratamento de exceções, o que dificulta a verificação de aplicações desenvolvidas, em C++ (e.g., programas que dependem do conjunto de bibliotecas *Standard Template Libraries* – STL) [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]. O modelo operacional proposto nesse trabalho, propõe que através da utilização de tipos abstratos de dados seja possível a verificação das bibliotecas STL, e não a utilização da própria STL. Blanc et al. expõe que a verificação do modelo operacional e suficiente para a verificação da corretude dos programas, tendo em vista que a prova das pré-condições, a partir de determinadas operações do modelo, são o suficiente para que se tenha a garantia das referidas pré-condições que são especificadas pela linguagem alvo das operações.

Essa abordagem mostra-se eficiente na busca por erros triviais em programas escritos em C++, no entanto, apresenta, encontra dificuldades quando são realizadas buscas mais profundas de erros, isso ocorre com frequência quando trata-se da modelagem interna dos métodos inerentes a linguagem alvo. No presente trabalho, esses problemas são contornados, com a simulação dos comportamentos que são utilizados por determinadas funções e métodos.

6. Conclusões

O presente trabalho propôs, utilizar um modelo operacional simplificado para verificar programas *multi-threads* baseados em programas desenvolvidos no *framework* multiplataforma Qt, incluindo precondições, com o intuito de verificar as principais propriedades relacionadas a *multi-threads* existentes no *framework* referenciado, verificando também uma aplicação em Qt e métodos implementados separadamente, em

seguida, os método e *benchmarks* foram integrados ao $ESBMC^{QtOM}$, expandindo assim a cobertura do referenciado *framework*. É importante ressaltar que o $ESBMC^{QtOM}$ antes do trabalho realizado não possuía nem um suporte para verificação de tais estruturas multi-threads.

Um modulo do framework multiplataforma Qt foi abordado: QtCore, incluindo duas bibliotecas, Qthreas e Qmutex. Além disso foram desenvolvidas duas suítes de testes automatizadas, com o intuito de validar as implementações desenvolvidas.

Os resultados experimentais demonstram a eficiência da utilização dessa abordagem para a verificação dos programas trabalhados na presente pesquisa, considerando-se o conjunto de testes desenvolvidos pode-se obter uma taxa de verificações corretas de 80%, dos casos de testes produzidos. Como trabalhos futuros a fim de aumentar a cobertura do modelo operacional simplificado *multi-threads,* mais classes e bibliotecas serão inseridas no modelo com o intuito de aumentar a sua cobertura.

7. Referências

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

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

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

National Aeronautics and Space Administration (2007). Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/. Acesso em: Setembro, 2016

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

Mandelbrot Example (2016a). Disponível em:<http://doc.qt.io/qt-5/qtcore-threads-mandelbrot-example.html.Agosto, 2016.

Monteiro, F. Verificação de Programas C++ Baseados no Framework Multiplataforma Qt. Manaus, pages 1-4.

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

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