SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2017; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Bounded Model Checking of C++ Programs based on the Qt Cross-Platform Framework

Felipe R. Monteiro[1*†], Mário A. P. Garcia[1],
Lucas C. Cordeiro[1,2], and Eddie B. de Lima Filho[1]

[1] *Faculty of Technology, Federal University of Amazonas, 69077-000, Brazil*
[2] *Department of Computer Science, University of Oxford, OX1 3QD, United Kingdom*

SUMMARY

The software development process for embedded systems is getting faster and faster, which generally incurs an increase in the associated complexity. As a consequence, technology companies tend to invest in fast and automatic verification mechanisms, in order to create robust systems and reduce product recall rates. In addition, further development-time reduction and system robustness can be achieved through cross-platform frameworks, such as Qt, which favor the reliable port of software stacks to different devices. Based on that, the present paper proposes a simplified version of the Qt framework, which is integrated into a checker based on satisfiability modulo theories (SMT), known as the Efficient SMT-based Context-Bounded Model Checker (ESBMC++), for verifying actual Qt-based applications, with a success rate of 89%, for the developed benchmark suite. Furthermore, the simplified version of the Qt framework, named as Qt Operational Model (QtOM), was also evaluated using other state-of-the-art verifiers for C++ programs. In fact, QtOM was combined with two different verification approaches: explicit-state model checking and also symbolic (bounded) model checking, during the experimental evaluation, which highlights its flexibility. The proposed methodology is the first one to formally verify Qt-based applications, which has the potential to devise new directions for software verification of portable code. Copyright © 2017 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Software Engineering; Qt Framework; Bounded Model Checking; Formal Verification;

1. INTRODUCTION

The current dissemination increase regarding embedded systems, aligned with the evolution of the associated software and hardware components, is an evidence of their major importance. In fact, they are becoming more robust and complex and now require processors with several cores and scalable shared-memory, among other advanced features, in order to attend the increased request for computational power, which can benefit from the combination of widespread programming languages and frameworks. In this scenario, Qt emerges as a powerful cross-platform framework for device creation and user interface/application development [1]. Particularly, Qt is a software development framework able to run on different platforms, as a native application, with few (or none) changes in the underlying code; indeed, it is broadly used by foremost companies (*e.g.*, LG Electronics [2], Sky [3], and AMD [4]), in order to build embedded devices to a wide range of applications, such as automotive, medical, infotainment, wearables, and automation. However, as the complexity of such systems grows, the user dependence on their proper operation is

†felipemonteiro@ufam.edu.br
*Correspondence to: Electronic and Information Research Centre at Federal University of Amazonas – Manaus, Amazonas, 69077-000, Brazil

rapidly rising. As a consequence, the reliability of embedded systems becomes a key point in the development process of many different commercial and application-specific devices.

Technology companies, such as Intel [5, 6], increasingly invest effort and time to develop fast and cheap verification alternatives, in order to verify correctness in their systems and then avoid financial losses [7, 8]. Among such alternatives, one of the most effective and less expensive ways is the model checking approach [9]; however, despite its advantages, there are many systems that could not be automatically verified, due to the unavailability of verifiers that support certain types of languages and frameworks. For instance, Java PathFinder is able to verify Java code, based on byte-code [10], but it can only support verification of Java applications that rely on the Android operating system if an abstract representation of the associated libraries, called operational model (OM), which conservatively approximates their semantics is available [11, 12].

The present work addresses the problem described above, identifies the main features of the Qt framework and, based on that, proposes an OM, which provides a way to analyse and check properties related to those same features. The developed algorithms were integrated into a bounded model checking (BMC) tool based on satisfiability modulo theories (SMT), known as the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) [13], in order to verify specific properties in Qt/C++ programs. The combination among ESBMC++ and OMs has been previously applied to support C++ programs, as described by Ramalho *et al.* [13]; however, in the proposed methodology, an OM is used to identify elements of the Qt framework and verify specific properties related to such structures, via pre and postconditions.

Contributions. The present paper extends a previous published work by Monteiro *et al.* [14] and Garcia *et al.* [15]. The respective OM was expanded, in order to include new features from the main Qt modules: Qt GUI and QtCore. Indeed, the main contributions here are *(i)* the support for sequential and associative template-based containers, *(ii)* the integration of QtOM into the verification process of the state-of-the-art C++ verifies DiVinE [16] and LLBMC [17], and *(iii)* the verification of two Qt-based applications known as *Locomaps* [18] and *GeoMessage* [19], respectively. In particular, representations for all libraries related to Qt container classes were included. Furthermore, the employed benchmark suite was extensively revised and expanded, in comparison with the mentioned previous work, and the performance of three SMT solvers (Z3 [20], Yices [21], and Boolector [22]) was evaluated, along with the proposed approach.

Finally, given the current knowledge in software verification, there is no other model checker that applies BMC techniques to verify Qt-based applications.

Availability of Data and Tools. The performed experiments are based on a set of publicly available benchmarks. All test modules, tools, and results, associated with the current evaluation, are available on a supplementary web page[†].

Outline. In section 2, the related work is discussed. Section 3, in turn, presents a brief introduction to the ESBMC++ architecture and satisfiability modulo theories, and also an abridgement of the Qt cross-platform framework. Section 4 describes a simplified representation of the Qt libraries, named as Qt Operational Model (QtOM), which also addresses pre and postconditions, while the formal implementation of sequential and associative Qt containers is detailed in section 5. In section 6, experimental results are presented, using several Qt/C++ benchmarks and also two Qt-based applications, where the first one demonstrates satellite, terrain, street maps, and Tiled Map Service (TMS) panning, among other features [18], and the second one generates User Datagram Protocol (UDP) broadcast based on XML files. Finally, conclusions and future work are tackled in section 7.

2. RELATED WORK

Given the current verification literature, there is no other available verifier that is able to check features of the Qt framework. As opposed, applications in many areas (*e.g.*, automotive, medical,

[†]http://esbmc.org/qtom

inflight entertainment, and laser technology) [1], which use such a framework, present several properties that must be verified, such as arithmetic under and overflow, pointer safety, array-bounds, correctness of containers usage, proper Qt framework method and function calling, and events handling, among others. Additionally, checking such properties through manual tests is an arduous and time-consuming process. In summary, BMC applied to software verification is a technique used in several verifiers [17, 23, 24, 25] and is becoming popular, mainly due to the rise of more sophisticated SMT solvers, which are based on efficient boolean satisfiability (SAT) solvers [20].

Merz, Falke, and Sinz [17] presented the Low-Level Bounded Model Checker (LLBMC), which is a bounded model checker for ANSI-C/C++ code. LLBMC also uses a low level virtual machine (LLVM) compiler, in order to convert ANSI-C/C++ programs into an LLVM intermediate representation and, based on that, it performs the required verification. Similarly to ESBMC++, Merz, Falke, and Sinz also apply SMT solvers to check verification conditions; however, differently from the approach present here, LLBMC does not support exception handling, which makes the verification of real programs, written in C++ (*e.g.*, programs based on Standard Template Libraries – STL), arduous. It is worth noticing that the LLVM intermediate representation loses some information about the structure of the respective C++ programs (*i.e.*, class relationship).

Barnat *et al.* introduced DiVinE [16], which is an explicit-state model checker for single- and multi-threaded ANSI-C/C++ programs. Its main purpose is to verify safety properties of asynchronous and shared-memory programs. DiVinE makes use of a compiler (CLang [26]) as front-end, in order to translate C++ programs into a LLVM intermediate representation, and then performs verification on the generated bitcode. Although DiVinE employs an implementation of the ANSI-C and C++ standard libraries, which allows proper verification of programs written in such languages, it does not possess any representation for the Qt framework.

Regarding static analysis approaches, I. Dillig, T. Dillig, and A. Aiken presented a fully automatic technique for static reasoning about unbounded data collections and arrays [27]. They analyse client code through extensions of uninterpreted functions, in order to model containers and map their contents. Such an approach is similar to the one present here, once they use indexed locations to keep on track the concrete data within sequential and associative containers; however, it does not offer full support to Qt Container classes.

Blanc, Groce, and Kroening described the verification of C++ programs that use STL containers, through predicate abstraction [28], with the use of abstract data types for STL usage verification, rather than the actual STL implementation and behaviour. Indeed, they show that correctness can be verified through OMs, by proving that preconditions on operations, in those same models, imply preconditions on standard libraries and postconditions may be as meaningful as the original ones. Such an approach is efficient in finding trivial errors in C++ programs, but it lacks a deeper search for bugs and misleading operations (*i.e.*, when they involve internal-method modeling). It is worth mentioning that, in the present work, behavioural simulations of certain methods and functions overcome the mentioned issue (see Section 4.2).

The C bounded model checker (CBMC) implements BMC for ANSI-C/C++ programs, through SAT/SMT solvers [23]. ESBMC was built on top of CBMC; therefore, they have similar verification processes. Indeed, CBMC can process programs using the goto-cc tool [29], which compiles source code into equivalent GOTO-programs (*i.e.*, control-flow graphs), using a GCC compliant style. From GOTO-programs, CBMC generates an abstract syntax tree (AST), which is then converted into an internal language-independent format used for the remaining steps. CBMC also uses two recursive functions $\mathcal{C}$ and $\mathcal{P}$ that compute the *constraints* (*i.e.*, assumptions and variable assignments) and the *properties* (*i.e.*, safety conditions and user-defined assertions), respectively. It automatically generates safety conditions that check under and overflow arithmetic, array-bounds violations, and `NULL`-pointer dereferences, in the spirit of Site's clean termination [30]. Finally, a verification condition generator (VCG) then derives verification conditions (VCs) from those formulae and sends them to a SAT/SMT solver. Although CBMC is supposed to perform deep verification on C++ programs, Ramalho et al. [13] and Merz et al. [31] reported that it failed when checking several simple C++ applications, which was also confirmed in this work (see section 6.4).

Indeed, QtOM is completely written in C++, which favors the integration within verification processes of other verifiers. In the present work, QtOM was not only integrated into ESBMC++, but also into DiVinE [16] and LLBMC [17], in order to perform a fairer evaluation.

3. BACKGROUND

The work presented here focuses on the verification of C++ programs based on the Qt framework, by using the ESBMC++ tool. ESBMC++ builds on the front-end of CBMC, in order to produce VCs for a certain C++/Qt program. Nonetheless, instead of passing VCs to a SAT solver, ESBMC++ encodes them through different background theories and then passes the associated results to an SMT solver. In this section, the ESBMC++ architecture is described and some Qt cross-platform framework features are explained.

3.1. ESBMC++

ESBMC++, whose architecture is shown in Fig. 1, is a context-bounded model checker based on SMT solvers, which is used for ANSI-C/C++ programs [13, 24, 32]. ESBMC++ verifies single- and multi-threaded programs and checks properties related to arithmetic under and overflow, division by zero, out-of-bounds index, pointer safety, deadlocks, and data races. In ESBMC++, the verification process is completely automated, *i.e.*, all process represented as gray boxes, in Fig. 1, do not require any effort from users to preprocess programs, at any stage.

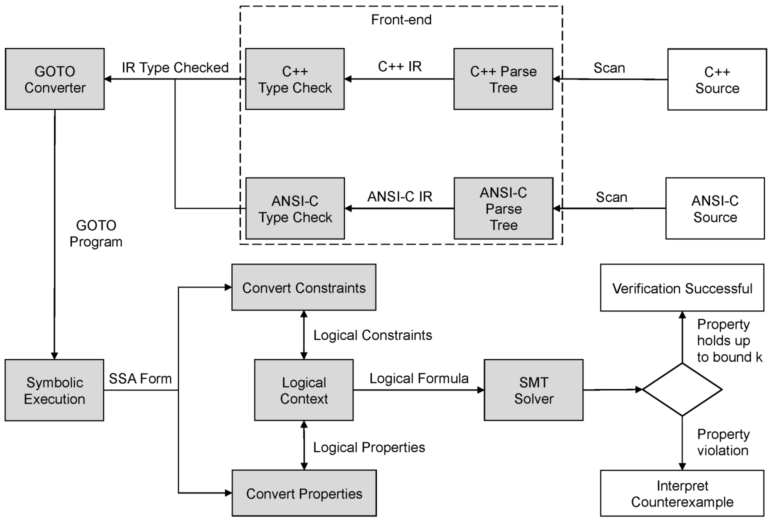


Figure 1. Overview of ESBMC++'s architecture.

During verification, the first step is the source code parser. Indeed, the parser used in ESBMC++ is heavily based on the GNU Compiler Collection (GCC), since such an approach enables ESBMC++ to find most of the syntax errors already reported by GCC. ANSI-C/C++/Qt programs are then converted into intermediate representation (IRep) trees, which are the basis for the remaining steps of the intended verification. Note that that OMs are the key for this conversion process, which will be explained in section 4. Thereafter, during the type-check step, an additional verification is performed, in IRep trees, which includes assignment, type-cast, pointer initialization, and function call checks, as well as template creation and instantiation [13].

Next, IRep trees are converted into goto expressions, which simplify statement representations (*e.g.*, replacement of *while* by *if* and *goto* statements) and are symbolically executed by the *GOTO-symex*; as a result, a Single Static Assignment (SSA) form is generated. Based on that, ESBMC++ computes two formulae, through two recursive functions: *constraints* (*i.e.*, assumptions and variable assignments) and *properties* (*i.e.*, safety conditions and user-defined assertions). Those formulae accumulate control flow predicates of each program point, and use them to store constraints (formula $\mathcal{C}$) and properties (formula $\mathcal{P}$), so that they properly reflect the program's semantics. Subsequently, those two first-order logic formulae are checked by an SMT solver.

Finally, if a property violation is found, a counterexample is provided by ESBMC++, which then assigns values to program variables, in order to reproduce the respective error. Indeed, counterexamples are of paramount importance for program-execution analysis and diagnosis, given that violations can be systematically traced.

3.2. *Satisfiability Modulo Theories (SMT)*

SMT solvers decide the satisfiability of first-order formulae, using a combination of different background theories, and thus generalize propositional satisfiability by supporting uninterpreted functions, linear and non-linear arithmetic, bit-vectors, tuples, arrays, and other decidable first-order theories. Given a theory $\mathcal{T}$ and a quantifier-free formula ψ, the latter is $\mathcal{T}$-satisfiable if and only if there is a structure that satisfies both formula and sentences of $\mathcal{T}$, that is, if $\mathcal{T} \cup \{\psi\}$ is satisfiable [33]. Given a set $\Gamma \cup \{\psi\}$ of formulae over $\mathcal{T}$, ψ is a $\mathcal{T}$-consequence of Γ ($\Gamma \models_{\mathcal{T}} \psi$) if and only if every model of $\mathcal{T} \cup \Gamma$ is also a model of ψ. This way, checking $\Gamma \models_{\mathcal{T}} \psi$ can be reduced to verifying the $\mathcal{T}$-satisfiability of $\Gamma \cup \{\neg\psi\}$.

The array theories of SMT solvers are typically based on McCarthy's axioms [34]. Function *select(a, i)* denotes the value of a, at index i, and *store(a, i, v)* denotes an array that is exactly the same as a, except that its value, at index i, is v. Formally, *select* and *store* can then be characterized by the axioms [20, 22, 35]

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

and

$$i \neq j \Rightarrow \mathit{select}\left(\mathit{store}\left(a, i, v\right), j\right) = \mathit{select}\left(a, j\right).$$

Tuples are used to model ANSI-C union and struct datatypes and provide the operations *store* and *select*, which are similar to those in arrays; however, they work on tuple elements. Each field of the tuple is represented by an integer constant; hence, the expression $\mathit{select}(t, f)$ denotes the field f of tuple t, while the expression $\mathit{store}(t, f, v)$ denotes a tuple t that, at field f, has value v. In order to check the satisfiability of a given formula, SMT solvers handle terms in the given background theory, using a decision procedure [36].

3.3. *The Qt Cross-Platform Framework*

Several software modules, known as frameworks, have been used for accelerating application development processes. In that context, the Qt cross-platform framework [1] represents a great example of a reusable set of classes, where software engineering was able to favor the development of graphical applications written in C++ [1] and Java [37]. It provides programs that run on different hardware/software platforms, with as few changes as possible, while maintaining the same power and speed. Panasonic Avionics [38], Zühlke [39], LG Electronics [2], Sky [3], and AMD [4] are some companies that apply Qt to the development of their applications [1].

According to the Cross-Platform Tool Benchmarking 2014 [40], Qt is the leading cross-platform application, user interface, and device development framework. Its libraries are organized into modules (see Fig. 2) that rely on the Qt Core module [1], which contains all non-graphical core classes. For instance, `QtCore` contains a set of libraries called Container Classes, which implement template-based container classes for general purpose, as an alternative for STL containers. In fact, such structures are widely known and applied to real Qt-based applications.

In addition to those submodules, `QtCore` contains the Qt Event System. In Qt, an event is represented by an object that inherits from the `QEvent` class (base class for the Qt Event System), which contains the necessary information about all actions (internal or external) related to an application. Once instantiated, this object is sent to an instance of the `QObject` class, which will call an appropriate handler method, based on its type.

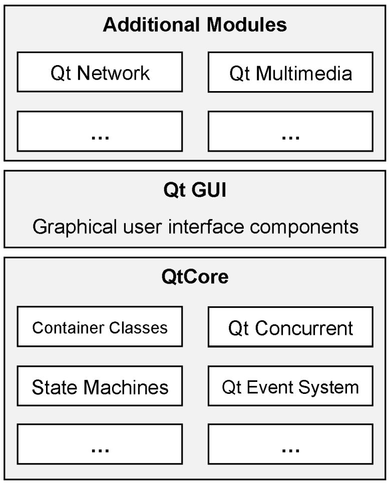
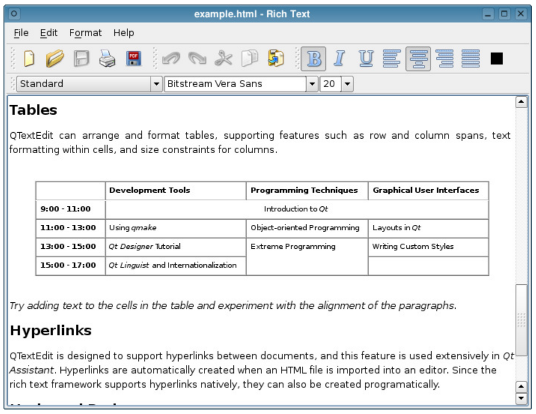


Figure 2. Brief overview of the Qt framework's structure.

Furthermore, the Qt framework provides a complete abstraction for the Graphical User Interface (GUI) part, by using native APIs, from different platforms, to query metrics and draw elements. It also provides signals and slots, in order to enable communication between objects [41]. Another important feature is the MetaObject compiler, which interprets programs and generates C++ code with meta information [42].

As an example of how Qt framework is used in software development, Fig. 3 shows Text Edit [43], a Qt application that demonstrates Qt's text engine editing facilities. Text Edit includes 32 Qt libraries (some are listed in Fig. 3b) and, from those, 90% belongs to the `QtGUI` and `QtCore` modules. As one may notice, not only the powerful features for GUI development are explored, but also Qt Containers (*e.g.*, `QMap`), file handling (*e.g.*, `QFile`), event handling (*e.g.*, and Qt Event System), among other features that certainly need to be considered during software testing as well. According to what was presented, one may notice that the complexity and robustness of programs based on the Qt framework directly affects verification procedures related to them, which is the focus of the present study.



Tables

QTextEdit can arrange and format tables, supporting features such as row and column spans, text formatting within cells, and size constraints for columns.

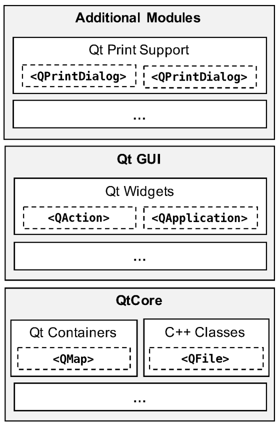
	Development Tools	Programming Techniques	Graphical User Interfaces
9:00 - 11:00	Introduction to *Qt*		
11:00 - 13:00	Using *qmake*	Object-oriented Programming	Layouts in *Qt*
13:00 - 15:00	*Qt Designer* Tutorial	Extreme Programming	Writing Custom Styles
15:00 - 17:00	*Qt Linguist* and Internationalization		

Try adding text to the cells in the table and experiment with the alignment of the paragraphs.

Hyperlinks

QTextEdit is designed to support hyperlinks between documents, and this feature is used extensively in *Qt Assistant*. Hyperlinks are automatically created when an HTML file is imported into an editor. Since the rich text framework supports hyperlinks natively, they can also be created programatically.

(a) Text Edit's GUI.



(b) Qt libraries used by Text Edit application.

Figure 3. Text Edit application example, which illustrates Qt's document-oriented rich text engine.

4. SMT-BASED BMC FOR C++ PROGRAMS BASED ON THE QT FRAMEWORK

Throughout the verification process with ESBMC++, the first step is the parser, as already mentioned (see Section 3.1), where ESBMC++ translates input code into an IRep tree, which encloses all information needed by the verification process. Thus, in order to accomplish this step, ESBMC++ must correctly identify each structure, in the respective program; however, ESBMC++ only supports verifying ANSI-C/C++ programs and, in spite of Qt code being written in C++, standard Qt libraries contain many hierarchical and very complex structures. As a consequence, the verification process for those libraries and their optimized implementations would unnecessarily complicate VCs and, in addition, they do not contain assertions to check specific properties, which results in an infeasible task.

In order to tackle that problem, the use of QtOM is proposed here, which is a simplified representation that considers the structure of each library and its associated classes, including attributes, method signatures, and function prototypes. There are also assertions integrated into QtOM, which ensure that each property is formally checked. Indeed, there are many properties to be verified, such as invalid memory access, negative time-period values, access to missing files, and null pointers, along with pre and postconditions, which are necessary to correctly execute Qt methods. In its current stage, QtOM does not support Qt events, since there is no abstract representation for the `QEvent` class yet. Additionally, it is worth noting that QtOM is manually connected to ESBMC++, right at the beginning of the verification process (see Fig. 4). This way, QtOM can assist the parser process to build a C++ intermediate representation, which contains all indispensable assertions for verifying the aforementioned properties. Finally, the remaining verification flow is carried out in the traditional way.

Comparing with previous works [14, 15], QtOM now encloses an initial representation for all libraries of the Qt Core and Qt GUI modules and also provides full support for all container classes, which are widely used in real applications.

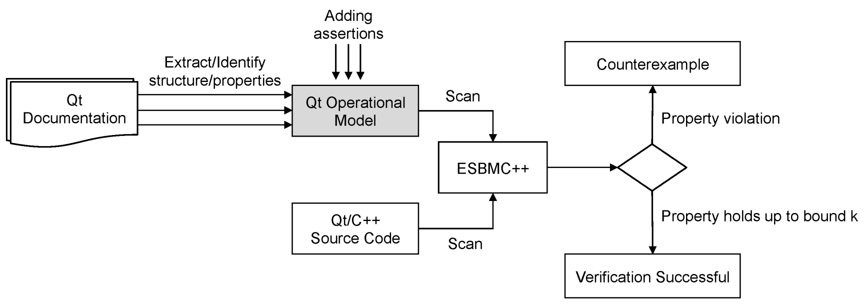


Figure 4. Connecting QtOM to ESBMC++.

4.1. *Preconditions*

From the use of simplified models, it is possible to decrease the complexity related to IRep trees. Consequently, ESBMC++ is able to build IRep trees that enclose all necessary properties for verifying the respective programs, in a much faster and low-complexity way. In addition, the use of assertions is indispensable for verifying properties related to methods from the Qt framework and also their execution, which is not covered by standard libraries. This way, such assertions are integrated into the respective methods, in order to detect violations related to the incorrect use of the Qt framework.

In summary, based on the addition of assertions, ESBMC++ is able to verify specific properties contained in QtOM. Besides, part of those properties can also be identified as preconditions, *i.e.*, properties that must hold, in order to allow proper execution of certain methods or functions. For instance, the proposed methodology can check if a parameter, which represents positions in arrays, is greater or equal to zero.

```
 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() +
12              `` is an empty message file'');
13   }
14   else
15   {
16     return NULL;
17   }
18 }
```

Figure 5. Code fragment of function `loadSimulationFile`, from the *GeoMessage* benchmark.

Fig. 5 shows a code fragment from a benchmark called *GeoMessage* Simulator, which provides messaging for applications and system components, in the ArcGIS platform [19]. As shown in this example, the method `setFileName()` handles a precondition (see line 6), where `m_inputFile` is an object of the `QFile` class, which provides an interface for reading from and writing to files [1]. Given that it sets a file name, the `QString` object `fileName` cannot be empty. This way, when `setFileName()` is called, ESBMC++ interprets its behaviour, as implemented by the QtOM.

```
 1 class QFile {
 2 ...
 3   QFile(const QString & name) { ... }
 4   ...
 5   void  setFileName(const QString & name){
 6     __ESBMC_assert(!name.isEmpty(),
 7           ``The string must not be empty'');
 8     __ESBMC_assert(!this->isOpen(),
 9           ``The file must be closed'');
10   }
11   ...
12 };
```

Figure 6. Operational model for the `setFileName()` method, from the `QFile` class.

Fig. 6 shows an OM corresponding to the `QFile` class, with an implementation of `setFileName()` (see lines 5-10), where only preconditions are checked. In particular, if the `QString` object, passed as a parameter, is not empty (see line 6), the associated operation is valid and, consequently, its assertion evaluates to *true*; otherwise, if an invalid operation is performed, such as an empty string passed as parameter, the assertion evaluates to *false*. In that case, ESBMC++ would return a counterexample, with all execution steps needed to reproduce such violation, in addition to the error described in the respective assertion.

From the software verification point of view, there are also methods/functions that do not present any property to be checked, such as the ones whose sole purpose is to print values on screen. Given that ESBMC++ performs software verification rather than hardware test, the verification process regarding correctness of the printed value was not addressed in this work. In summary, such methods present only method signatures (no body), so that the proposed verifier is able to recognize its structure, during the analysis process, in order to build a reliable IRep tree; however, they do not present any modelling (method body), since there is no property to be verified.

4.2. *Postconditions*

In real programs, there are methods that do not only contain properties that must be handled as preconditions, but also considered as postconditions [14, 15]. For instance, according to the Qt documentation [1], `setFileName` (see section 4.1) must not be called if the respective file has already been opened, which is checked, in its source code, by the *if* structure found on lines 3 and 4 (see Fig. 5).

Nevertheless, the execution of the instruction on line 4 (see Fig. 5) would be non-deterministic for ESBMC++, as well as the assertion on line 8 (see Fig. 6) of the associated OM, once it does not know if the file has been opened or not. Thus, it is clear that one needs to simulate the behaviour of the `isOpen()` method, in order to consistently verify properties related to file manipulation. As the `QFile` class indirectly inherits the methods `open()` and `isOpen()`, from the `QIODevice` class, behavioural simulations of those methods were built into the `QIODevice` OM, as shown in Fig. 7.

```
class QIODevice {
    ...
    bool QIODevice::open(OpenMode mode){
        this->__openMode = mode;
        if (this->__openMode == NotOpen)
            this->__isOpen = false;
        this->__isOpen = true;
    }

    bool  isOpen() const{
        return this->__isOpen;
    }
    ...
private:
    bool __isOpen;
    OpenMode __openMode;
    ...
};
```

Figure 7. Operational model for the `open()` and `isOpen()` methods, in the `QIODevice` class.

Therefore, all methods that result in a condition that must hold, in order to allow proper execution of future instructions, must present a behavioural simulation. As a consequence, a given OM must strictly follow the specification described in the official documentation [1] and guarantee the same behaviour, as well as include all necessary tools for code verification. As a result, one needs to check the equivalence between operational model and original library, with the goal of ensuring the same behaviour, having in mind that operational models are a simplification of the original libraries, with all tools necessary for code verification.

5. OPERATIONAL MODELS FOR CONTAINERS

The Qt Core module provides a set of template-based container classes, as an alternative to STL containers from C++ [1]. For instance, if the developer needs a resizable stack of `QWidgets`, an alternative would be `QStack<QWidget>`. In addition, such containers also use Java- and STL-style iterators, in order to move across stored items.

In particular, those container classes can be classified into two subgroups: sequential and associative, depending on the storage structure of each one. Classes `QList`, `QLinkedList`, `QVector`, `QStack`, and `QQueue` belong to the sequential group, while classes `QMap`, `QMultiMap`, `QHash`, `QMultiHash`, and `QSet` belong to the associative group.

As a consequence, a core language, with the goal of formalising the implementation of each container class, is defined, in section 5.1. Then, in sections 5.2 and 5.3, formal implementations, for sequential and associative containers, respectively, are described.

5.1. *Language*

In order to implement OMs for Qt containers, the core container language formalisation described by Ramalho *et al.* [13] is used, which extends the translation formulae $\mathcal{C}$ and $\mathcal{P}$. In fact, that core language is adapted, in order to properly formalise the verification of both sequential and associative Qt containers, as shown in Fig. 8.

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

Figure 8. Core container syntax for QtOM.

Here, the base elements are split into two syntactic domains: V for values and K for keys. Nonetheless, the remaining domains I, P, $\mathbb{N}$, and C are kept for iterators, pointers, integer indices, and proper container expressions, respectively. Thus, the variables k and v, of type K and V, respectively, are added. In addition, the notation I_v represents a value stored in an underlying container, at a position pointed by the iterator I, and I_k represents a key. Indeed, such notations are abbreviations for $store(i, I_{pos}, I_v)$ and $store(i, I_{pos}, I_k)$, respectively, where the expression $store(t, f, v)$ denotes a tuple t that, at field f, has value v. Similarly, P_k and P_v represent key and value, respectively, at position P.

Additionally, three other methods are included, as follows. $C.insert(k, v)$ inserts an element into a container, with key k and value v, and returns an iterator, which points to the new element, whose position depends on the container's type. $C.search(k)$ returns an iterator, which points to the first evidence of an element, with a corresponding key k. Similarly, $C.search(v)$ returns an iterator, which points to the first evidence of an element, with corresponding value v. Nonetheless, in both methods, if there is no corresponding key or value, they return $C.end()$. Finally, C_k is a memory address that stores the beginning of key containers, as well as C_v is used for value containers.

All remaining elements, from the mentioned core language, are used here, exactly as described by Ramalho *et al.* [13].

5.2. *Sequential Containers*

Qt sequential containers are built into a structure to store elements, in a certain sequential order [44]. According to the Qt documentation, `QList` is the most commonly used container class and implements a list structure, in order to store values that can be accessed by index. Similarly, `QLinkedList` also implements a list structure, although it uses iterators rather than integer indexes. `QVector` corresponds to a resizable array structure and, finally, `QStack` and `QQueue` provide structures that implement *last in, first out* and *first in, first out* policies, respectively.

To properly simulate sequential containers, the proposed models make use of the core language, which was described in section 5.1. As can be noticed in Fig. 9, sequential containers are implemented through a C_v pointer, for container values, and also with a C_{size}, which is used to represent the size of the container (where $C_{size} \in \mathbb{N}$). In addition, iterators are modeled through two variables: one of type $\mathbb{N}$, which is called i_{pos} and contains the index value pointed by an iterator, and another of type P, which is called I_v and points to the underlying container.

Note that all methods, from those libraries, can be expressed as simplified variations of three main operations: insertion ($C.insert(I, V, \mathbb{N})$), deletion ($C.erase(I)$), and search ($C.search(V)$). As part of the SSA transformation, side effects on iterators and containers are made explicit, so that operations return new iterators and containers.

For instance, a container c with a call *c.search*(v) is considered, which performs a search for an element v, in the container. Then, if such an element is found, it returns an iterator that points to the respective element; otherwise, it returns an iterator that points to the position immediately after the last container's element (*i.e.*, $c.end()$). In that case, the statement "$c.search(v)$;" becomes "$(c', i') = c.search(v)$;", which has explicit side effects. Thus, the translation function $\mathcal{C}$ describes

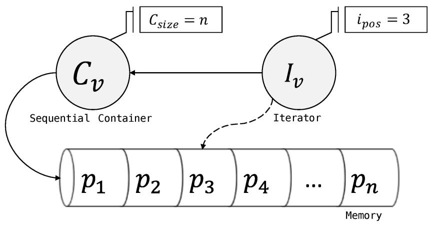


Figure 9. Sequential container model.

constraints relating the "before" and "after" versions of the respective model variables. Indeed, notations with apostrophe (*e.g.*, c' and i') represent the state of model variables, after the respective operation, and simplified notations (*e.g.*, c and i) represent their previous states. Furthermore, $select(c, i = lower_{bound} \dots i = upper_{bound})$ represents a loop expression (such as for and $while$), where each value of c, from $lower_{bound}$ to $upper_{bound}$ positions, will be selected. Similarly, $store(c_1, lower^1_{bound}, select(c_2, lower^2_{bound})) \dots store(c_1, upper^1_{bound}, select(c_2, upper^2_{bound}))$ also represents a loop expression, where each value of c_2, from $lower^2_{bound}$ to $upper^2_{bound}$ positions, will be stored into c_1, in the $lower^1_{bound}$ to $upper^1_{bound}$ positions, respectively. Hence,

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

Regarding sequential containers, methods $C.insert(I, V, \mathbb{N})$ and $C.erase(I)$ behave as described by Ramalho *et al.* [13].

5.3. *Associative Containers*

The associative container group contains five classes: `QMap`, `QMultiMap`, `QHash`, `QMultiHash`, and `QSet`. `QMap` implements an associative array, which connects each key, of a certain type K, to a value, of a certain type V, where associated keys are stored in order. On the one hand, `QHash` presents a behaviour similar to that of `QMap`; however, data is stored in an arbitrary order. On the other hand, `QMultiMap` and `QMultiHash` represent subclasses of `QMap` and `QHash`, respectively; however, both specify interfaces where one key can be associated with multiple values. Finally, a single-valued mathematical set is implemented by `QSet`.

In order to implement associative containers, a C_v pointer, for the container's values, and a C_{size}, for the size of the container, are also used; however, a C_k pointer is employed for the container's key, as shown in Fig. 10. Particularly, C_k and C_v are connected by an index, *i.e.*, given a container c that contains a key k and a value v, it is assumed that

$$[\forall \omega \in \mathbb{N} | 0 \le \omega < C_{size}]$$

and

$$k \to v \iff select\,(C_k, \omega) = k \wedge select\,(C_v, \omega) = v,$$

where $(k \to v)$ denotes that a key k is associated with a value v and ω represents a valid position in C_k and C_v. Furthermore, function *select(a, i)* denotes the value of a, at index i [13]. Once again, all operations in those libraries can be expressed as a simplified variation of the three main ones cited in section 5.2.

Therefore, the insertion operation for associative containers can be performed in two different ways. Firstly, if the order does not matter, a new element is added at the end of C_k and C_v. This way, given a container c, the method call $c.insert(k, v)$ inserts, in the container c, the value v associated with the key k; however, if k already exists, it replaces the value associated with k by v and returns

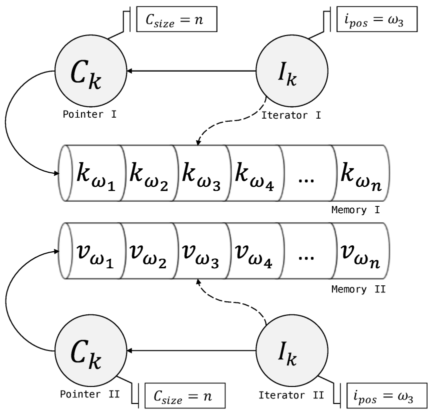

$C_{size} = n$
$i_{pos} = \omega_3$
C_k
I_k
Pointer I
Iterator I
k_{ω_1} k_{ω_2} k_{ω_3} k_{ω_4} ... k_{ω_n}
Memory I
v_{ω_1} v_{ω_2} v_{ω_3} v_{ω_4} ... v_{ω_n}
Memory II
C_k
I_k
Pointer II
$C_{size} = n$
Iterator II
$i_{pos} = \omega_3$

Figure 10. Associative container model.

an iterator that points to the inserted/modified element. Thus,

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

There is another version of the insert method, where the order of keys matters. This way, all variables cited above are considered and a comparison is added, in order to ensure that the new element is inserted into the right order. Thus,

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

In cases where keys with multiple associated values are allowed, the comparison to check if there already is an element with the respective key is ignored. Finally, in order to accomplish a deletion, the erase method is implemented, which is represented by $erase(i)$, where i is an iterator that points to the element to be deleted. It removes the element pointed by i, shifting backwards all elements followed by that. Thus

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

One can notice that such models implicitly induce two main properties, in order to properly perform the respective operations. Firstly, C_k and C_v are assumed to be non-empty (*i.e.*, that C_{size} holds), for search and erase operations. Secondly, i is considered as an iterator over the respective underlying container (*i.e.*, given a c container with C_k and C_v as base pointer, $I_k = C_k$ and $I_v = C_v$ holds). Indeed, these and other specific properties are handled in the respective operational models, as described in section 4.

5.4. *QtOM Correctness*

The original idea of performing model checking of real embedded applications, through an environment operational model, has already been done before in the literature [11, 12, 14, 15, 45], and the correctness of such operational model is actually a major issue. Consequently, the usefulness of QtOM relies on the fact that it correctly represents the original Qt libraries. In that sense, all developed Qt modules were manually verified and exhaustively compared to the original ones, in order to ensure the same behaviour. In summary, they all contain pre and postconditions, with the goal to ensure that a (given) predicate holds before and after the execution of a (given) function, respectively.

Additionally, although QtOM is a new implementation, it consists in (reliably) constructing a simplified model of the related Qt libraries, using the same language and by means of the original code and documentation (pre and postconditions are tested using assertions within the code itself), which thus tends to decrease the resulting number of errors. One may notice further that the behaviour of the Qt libraries functions are actually represented in C/C++ programming languages, using native functions (*e.g.*, *malloc*, *free*, *assert*). The soundness proof for those native functions, which are already supported by ESBMC, can be found in Cordeiro *et al.* [24].

Although proofs regarding the soundness of the entire QtOM could be carried out, it consists in a hard task, since it depends on many different factors (*e.g.*, memory model). In order to further improve the correctness of QtOM, conformance testing was also employed, similarly to Cámara *et al.* [46]. The basic idea is to compare the behaviour of standard Qt libraries to QtOM, with the goal of measuring their similarity. Initially, a test suite named *esbmc–qt* was developed, which contains a set of 711 benchmarks written in C++/Qt (*cf.* Section 6.1). As a first step, an exhausted manual analysis was performed, in order to ensure that every benchmark had assertions to check a given property (*e.g.*, invalid memory access and containers usage) and to properly identify its expected behaviour. Each benchmark containing a failed assertion was assigned as FAILED; otherwise, it was assigned as SUCCESSFUL. Then, all benchmarks from *esbmc–qt* were compiled and executed using QMake tool $v3.0$ [1], which automatically generated a Makefile to properly compile a Qt program with GCC $v4.9.4$ and the Qt framework $v5.2.1$ (*i.e.*, standard Qt libraries) [1]. Importantly, all results from the aforementioned manual analysis were compared with the behavior of each benchmark, in order to ensure that it executed as expected. As a result, such behaviour is defined as a reference point, as shown in Fig. 11, in the comparison against QtOM.

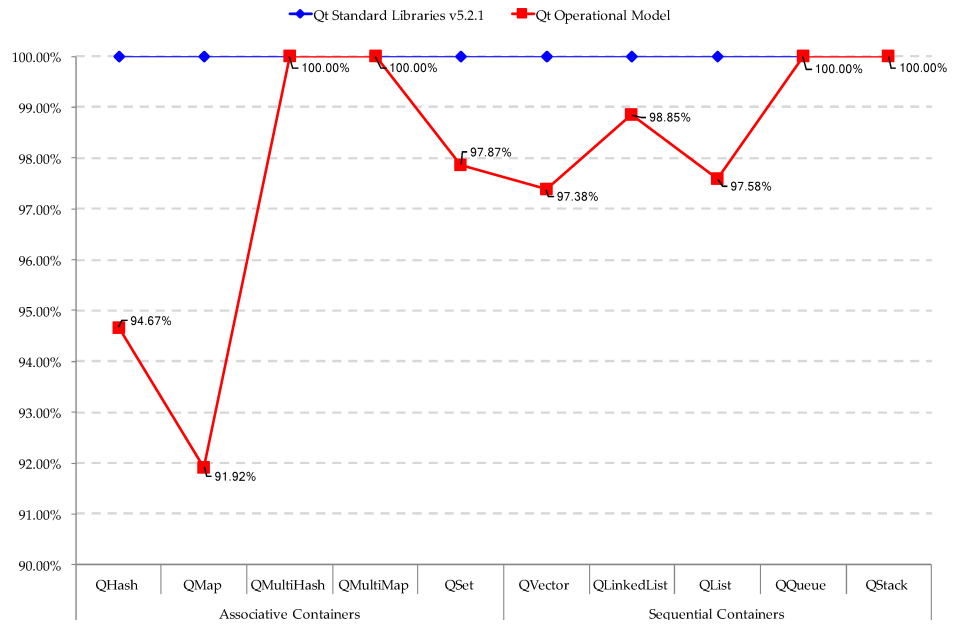


Figure 11. Similarity evaluation between the standard Qt libraries and QtOM.

Lastly, QtOM replaced Qt standard libraries in the compilation process of all benchmarks from *esbmc–qt*. The same compiler was applied (*i.e.*, GCC) and all benchmarks were successfully executed. Nonetheless, as one may noticed in Fig. 11, a few benchmarks did not present the exact same behaviour of the original Qt framework. More specifically, an overall similarity of 97.83% was identified. On the one hand, the benchmark sets `QMultiHash`, `QMultiMap`, `QQueue`, and `QStack` did not present any unexpected behaviour and obtained 100% of similarity. On the other hand, the benchmark sets `QHash`, `QMap`, `QSet`, `QVector`, `QLinkedList`, and `QList` presented minor variations and obtained 94.67%, 91.92%, 97.87%, 97.38%, 98.85%, and 97.58% of similarity, respectively. Such variations were due to the QtOM's implementation of iterators, which does not cover all achievable states. Indeed, such behaviour is explained in details in Section 6.4, once it directly affected the verification result in some particular cases. In addition, exceptionally for `QHash`, `QSet`, and `QVector` benchmark sets, a minor portion of behaviour variation was caused by the fact that some functions, such as `squeeze()` and `reserve()`, were not properly implemented, once they deal with fine tuning memory usage and QtOM does not have any internal representation for that. In order to deal with such behaviour variations, a more refined implementation for iterators and an internal memory model for representing memory usage along execution should be considered.

6. EXPERIMENTAL EVALUATION

This section is split into three parts. Section 6.1 describes the experimental setup and all benchmarks used for performing evaluations. In section 6.2, the correctness and also the performance of the proposed method are verified, using standard single-threaded C++/Qt programs, which were mostly extracted from the Qt documentation [1]. Finally, verification results for two real applications through the proposed QtOM, named as *Locomaps* [18] and *GeoMessage* Simulator [19], are described in section 6.5.

6.1. Experimental Setup

In order to evaluate the effectiveness of the proposed approach, regarding the verification of Qt programs, a set of benchmarks, named *esbmc–qt*, was built. In summary, it contains all benchmarks used in the present evaluation, with 711 Qt/C++ programs (12,903 code lines).

The mentioned benchmarks are split into ten main suites: `QHash`, `QLinkedList`, `QList`, `QMap`, `QMultiHash`, `QMultiMap`, `QQueue`, `QSet`, `QStack`, and `QVector`. Each suite contains benchmarks for the respective container classes, which access the Qt Core and the Qt GUI modules. Some of those benchmarks were directly taken from the Qt documentation, and the remaining ones were specifically developed for testing all features provided by the Qt framework. Importantly, even though some benchmarks do not implement real Qt-based applications, they are still valuable to analyse to which extent QtOM is able to handle and detect (known) errors.

Each benchmark is manually tested prior to being added to its respective suite. By means of those manual tests, one is able to identify whether a given benchmark does or does not have any bugs and complies with the intended operation. Thus, based on such inspection, it was possible to ensure that 353 out of the 711 benchmarks contain bugs (*i.e.*, 49.65%) and 358 are flawless (*i.e.*, 50.35%). Indeed, such kind of inspection is essential for the experimental evaluation, once it enables the comparison of verification results from each model checker and properly evaluates whether real errors were found. Particularly, the buggy portion of benchmarks present 5 errors sources:

- **Invalid memory access.** Arrays, objects, or pointers that attempt to access invalid memory addresses.
- **Time-period values.** Use of invalid time parameters in time-period specifications, which are key to the execution of some Qt features, such as those offered by the `QTime` class.
- **Access to missing files.** Invalid access or manipulation of all files, in a given program, which are handled by a set of Qt libraries (*e.g.*, `QIODevice` and `QFile`).
- **Null pointers.** Use of pointers that do not refer to any object or function (*i.e.*, `NULL` pointers) in invalid operations.
- **String manipulation.** Incorrect manipulation of unicode character string representations, which are provided by the `QString` class.
- **Container usage.** Incorrect manipulation of template-based container classes (*e.g.*, erase data from empty collections).

All experiments were conducted on an otherwise idle Intel Core i7-4790, with 3.60 GHz clock and 24 GB (22 GB of RAM and 2 GB of swap space), running Fedora OS (64 bits) and ESBMC++ 1.25.4, with three different solvers: Z3 v4.0, Boolector v2.0.1, and Yices 2 v4.1. The time and memory limits, for each benchmark, were set to 600 seconds and 22 GB, respectively. In addition, an evaluation using CBMC v5.1, LLBMC v2013.1, and DiVinE v3.3.2, combined with QtOM, was performed, in order to provide comparisons against ESBMC++. The indicated time periods were measured using the `clock_gettime` function from the `time.h` library [47].

6.2. Comparison Regarding SMT Solvers

It is widely known that different SMT solvers can heavily affect the obtained results, given that there is no homogeneity regarding implementation approaches and supported logics [24]. Firstly, verification using the three mentioned SMT solvers were performed: Z3, Boolector, and Yices. In order to perform such comparison, ESBMC++ was combined with each SMT solver to verify all benchmarks. As shown in Fig. 12, the results are evaluated by means of coverage (*i.e.*, percentage of correct verifications) and verification time. On the one hand, the worst performance was with Yices, which presented a coverage of approximately 57%, in 11.38 minutes. On the other hand, Boolector and Z3 presented a coverage rate of approximately 89% and 83%, respectively; however, the verification process with Z3 took 19.60 minutes to be concluded, while Boolector needed 16.08 minutes (*i.e.*, Boolector was approximately 1.2x faster than Z3, with higher accuracy). Furthermore, Yices presented the lowest coverage and time, because it does not support tuples; hence, it cannot properly solve the SMT formulae originated from the verification process of several benchmarks. In

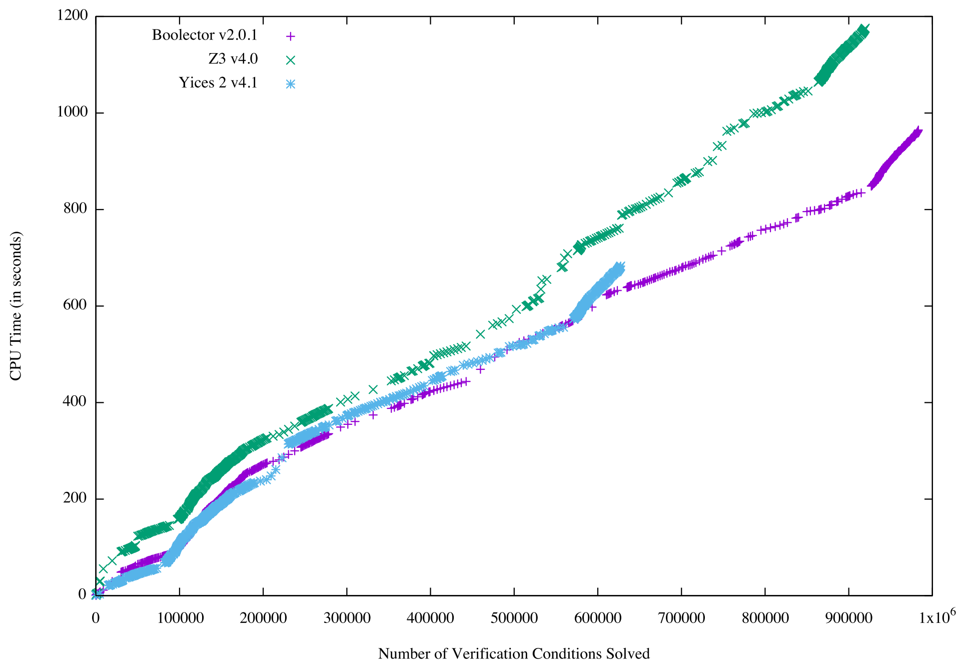


Figure 12. Comparison among SMT solvers.

summary, as shown in Fig. 12, Boolector presents itself as a better fit for the proposed verification process.

6.3. *Comparison Against Model Checkers*

All benchmarks, in the *esbmc-qt* suite, were automatically verified by ESBMC++, in order to check its correctness and efficiency. Besides the aforementioned comparison of SMT solvers, an evaluation about the performance of distinct model checkers was also carried out. Indeed, as already mentioned, there is no known checker for the Qt framework, neither an OM similar to the one proposed here (QtOM), using the C++ language. Nonetheless, due to the QtOM versatility, it is also possible to connect it to the verification processes of LLBMC [31] and DIVINE [16], as shown in Fig. 13, whose basis is the translation of source code into an LLVM intermediate representation (LLVM IR), also referred as bitcode. In fact, LLVM is a Static Single Assignment (SSA) based representation widely used by verifiers, due to its low-level nature, which can facilitate program analysis [16, 31].

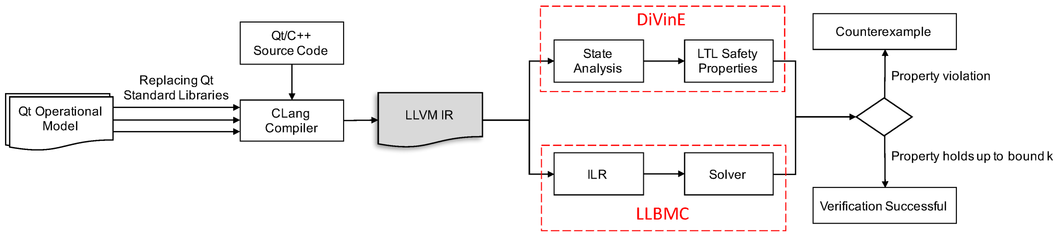


Figure 13. Connecting QtOM to DiVinE and LLBMC.

Initially, DiVinE and LLBMC precompile the source code using CLang [26], in order to generate an LLVM bitcode. At this level, QtOM was integrated, as a stub into their compilation processes, such that bitcode containing source code and QtOM information was produced. Henceforth, both model checkers carry out the verification process differently. On the one hand, LLBMC processes bitcode (*i.e.*, unrolls loops and inline-expands functions), which is then encoded into the LLBMC's intermediate logic representation (ILR). After simplifications, the ILR formula is translated into an SMT formula and finally passed to an SMT solver [31]. On the other hand, DiVinE directly interprets bitcode, in order to create the state space to be explored. Besides, DiVinE specifies safety properties using Linear Temporal Logic (LTL) [16].

To this end, a comparison regarding the performance of LLBMC and ESBMC++, which are SMT-based BMC model checkers, and DIVINE, which employs explicit-state model checking, was carried out. In addition, the initial intention was to also carry out comparisons with CBMC [23]; however, even connected to QtOM, it was not able to carry out the verification of several benchmarks, as already reported by Ramalho et al. [13] and Merz et al. [31], which caused its removal from the evaluation process.

6.4. *Verification Results for the Developed Benchmark Suite*

The chosen tools were executed using three scripts: one for ESBMC++, which reads its parameters from a file and provides execution[‡], another for LLBMC, which first compiles programs to *bitcode*, using CLang [§] [26], and then reads parameters from a file and runs such tool[¶], and a third one for DiVinE, which also first precompiles C++ programs to *bitcode*[‖] and then performs verification on that[**]. The loop unrolling defined for each tool (that is, the <bound> value) depends on each benchmark. To that sense, each benchmark is manually analysed, in order to identify its highest upper bound of loop. As a consequence, its verification bound is set as the upper bound value plus one. For now, LLBMC does not support exception handling and *bitcode* was generated without exceptions (*i.e.*, with the *-fno-exceptions* flag of the compiler). If exception handling is enabled, then LLBMC always aborts the verification process.

Table I. Results of the comparison among ESBMC++ v1.25.4 (using Boolector as SMT solver), LLBMC v2013.1, and DiVinE v3.3.2.

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

Table I shows the experimental results for the combinations between QtOM and ESBMC++, using Boolector as SMT solver, LLBMC and DiVinE. Here, *TC* is the number of Qt/C++ programs, *L* is the total code lines, *Time* is the total verification time, *P* is the number of flawless benchmarks verified correctly (*i.e.*, correct positive results), *N* is the number of faulty benchmarks verified correctly (*i.e.*, correct negative results), *FP* is the number of false positive results (*i.e.*, the tool reports flawless incorrectly), *FN* is the number of false negative results (*i.e.*, the tool reports faulty

[‡]esbmc *.cpp --unwind <bound> --no-unwinding-assertions -I /home/libraries/ −−memlimit 14000000 −−timeout 600
[§]/usr/bin/clang++ -c -g -emit-llvm *.cpp -fno-exceptions
[¶]llbmc *.cpp –ignore-missing-function-bodies –max-loop-iterations=<bound> –no-max-loop-iterations-checks
[‖]divine compile –llvm -o main.bc *.cpp
[**]divine verify main.bc –max-time=600 –max-memory=14000 -d

incorrectly), and *Fail* is the number of internal errors (*e.g.*, parse errors), during verification. It is worth mentioning that ESBMC++ with Boolector does not run out of memory or time, in any suite.

As one can see, in Table I, only 1.1% of the benchmarks with ESBMC++ reported verification $Fail$, which occurred when it was not able to check a given program due to internal errors. DiVinE and LLBMC, in turn, presented $Fail$ rates of 6.9% and 3.4%, respectively, which occurred when such tools did not generate bitcode for a given program or the verification ran out of time or memory. Regarding FP results, DiVinE presented the best performance, followed by LLBMC and ESBMC++. However, ESBMC++ achieved the lowest rate for FN results, followed by DiVinE and LLBMC, which is due to how pointers are handled by each tool. Internally, QtOM represents iterators through pointers and arrays, in order to simulate their real behaviours (see Section 5); however, such structure does not cover all achievable states. Particularly, when a deletion is performed on a container that has more than one iterator pointing to it, all iterators pointing to positions after the one deleted are lost. This behaviour affects the postconditions of a program, which directly influence *FP* and *FN* results. Arrays and pointers have been extensively employed, in order to obtain simple structures (*i.e.*, without classes and structs in its representation), which decreases the complexity of the verification process (cf. Section 4.2). In summary, the combination between ESMBC++ and QtOM results in a robust checker; however, there still are some gaps to be filled, regarding the support of C++ language (*e.g.*, full support of template structures), as described by Ramalho *et al.* [13].

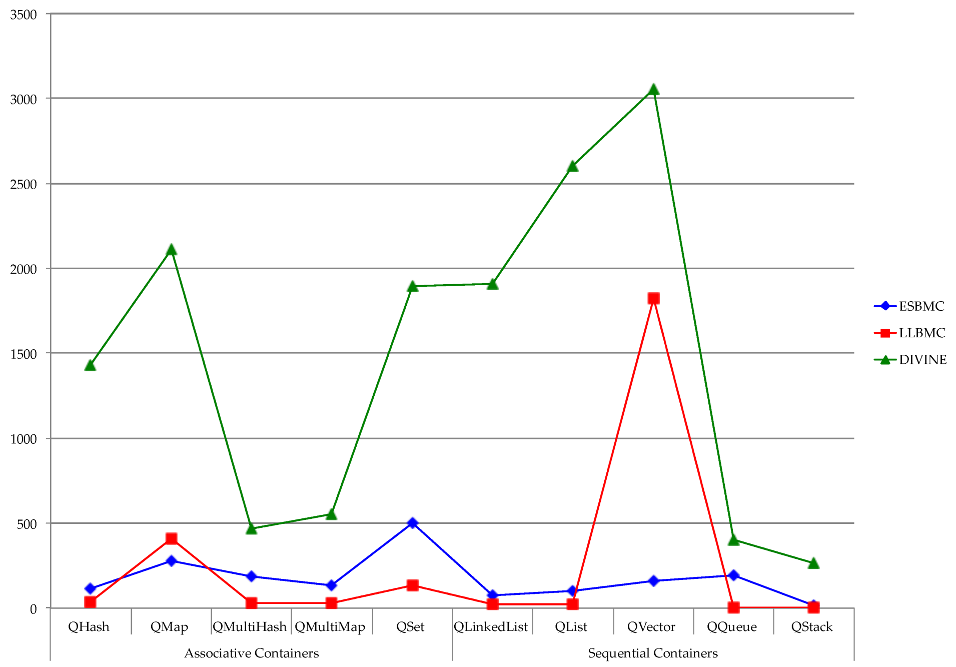


Figure 14. Comparison among verification times regarding ESBMC++, LLBMC and DiVinE.

Note that the more code lines a program has, the more structures it presents; therefore, its verification process becomes more complex. However, as one can see in Fig. 14, the suites `QMap` and `QSet` presented the largest verification times for ESBMC++, although `QVector` is the one with the greatest number of code lines. That occurred because not only the code line number matters, but also the amount of loops inside a program, which has a directly effect on verification times. In fact, the internal structures of OMs associated to `QMap` and `QSet` contain more loops than the others, which is the cause of the longer verification times. As one can see in Fig. 14 as well, LLBMC presented its largest verification time for the `QVector` benchmarks, which mainly occurred due to two benchmarks reporting $timeout$; however, LLBMC translation process with CLang only takes a few seconds on average (much faster than DiVinE) and, if the aforementioned benchmarks (*i.e.*, $timeout$

reports) were disregarded, LLBMC would be 1.3x faster than ESBMC++. Moreover, DiVinE is the slowest tool among the aforementioned ones, because its translation processes (C++ into bitcode) are 3x slower than actual verification. Due to that, test suites with more benchmarks will take longer to be checked by DiVinE, which is the case of `QVector`, `QList`, `QMap`, `QLinkedList` and `QSet`.

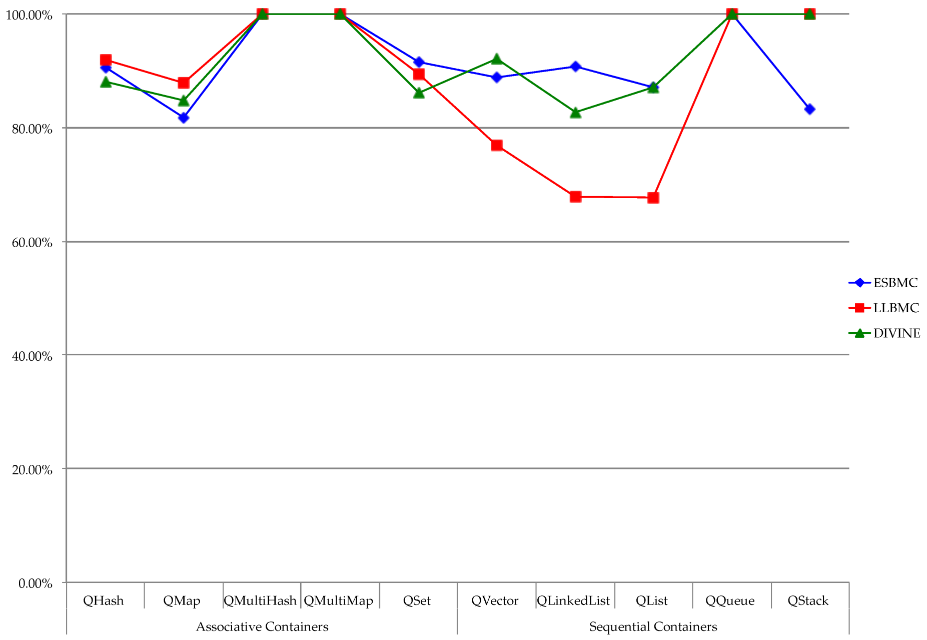


Figure 15. Comparison among coverage results regarding ESBMC++, LLBMC and DiVinE. .eps

As one can see in Fig. 15, all model checkers presented a coverage rate above 80% for associative containers; however, only LLBMC did not keep this same rate for sequential ones. All benchmarks from the `QMultiMap` and `QMultiHash` suites were correctly verified by the three model checkers. The `QHash`, `QMap`, and `QSet` suites, in turn, presented an average rate up to 6.7% for *FP* and *FN* results (*i.e.*, 3 to 18 out of 267 benchmarks), due to limitations regarding internal iterator representations. In addition, for LLBMC and DiVinE, 4.5% and 13.9% of the associative-container benchmarks, respectively, were not verified (*i.e.*, 12 and 37 out of 267 benchmarks, respectively), due to failures in the translation process of C++ programs into bitcode. With respect to sequential containers, LLBMC presented low coverage rates for the `QVector`, `QLinkedList`, and `QList` suites, that is, about 67.7% to 77% (*i.e.*, 84/117 out of 124/152 benchmarks, respectively), given that approximately 22.9% (*i.e.*, 83 out of 363 benchmarks) of the benchmarks presented *FN* results, due to the aforementioned problems (iterator representations). In addition, 5% (*i.e.*, 18 out of 363 benchmarks) of the benchmarks have not been verified, once LLBMC was not able to generate their bitcodes. ESBMC++ and DiVinE, in turn, presented a maximum error rate of 6.6% (*i.e.*, 24 out of 363 benchmarks) for the `QVector`, `QLinkedList`, and `QList` suites, due to errors in the postconditions analysis. Moreover, all benchmarks from the `QQueue` and `QStack` suites were correctly verified, except for two benchmarks of `QStack` suite verified by ESBMC++, once Boolector was not able to solve the SMT formula generated from them.

Furthermore, although the `QList`, `QMap`, `QVector`, and `QSet` suites reported more FP and FN test results with ESBMC++ than the other options, the coverage rate regarding benchmarks correctly verified, for each one, was around 80 to 90%, which demonstrates its effectiveness regarding verification, given that each benchmark checks a different feature from a different container.

It is also worth mentioning that ESBMC++ was able to detect 89.1% of the bugs in benchmarks (*i.e.*, 631 out of 708 benchmarks), which also demonstrates its effectiveness. Similarly, LLBMC and DiVinE presented an overall coverage of 81.4% and 88.7% (*i.e.*, 576 and 630 out of 708 benchmarks), respectively, which also demonstrates the suitability regarding the combination of QtOM with other checking tools. As a consequence, the proposed methodology is not limited to a given tool and may be adapted to specific applications, where some approaches are more suitable than others.

6.5. *Verification Results for Real Qt-based Applications*

Given that the proposed set of benchmarks is focused on verifying specific properties in Qt modules, it is interesting to include results for real applications, which were written with no concern regarding that. The following paragraphs describe the respective applications and the associated results.

The Qt sample application called *Locomaps* [18] demonstrates satellite, terrain, street maps, tiled map service planning, and Qt Geo GPS Integration, among other features. Through the same source code, such an application can be cross-compiled and run on Mac OS X, Linux, and Windows. It contains two classes and 115 Qt/C++ code lines, using five different APIs from the Qt framework: `QApplication`, `QCoreApplication`, `QDesktopWidget`, `QtDeclarative`, and `QMainWindow`. Note that Qt/C++ code from this application, the APIs, and the libraries used in the APIs are considered in the verification process, as well as properties related to them.

ArcGIS for Military [48] is a geography platform for creating, organizing, and sharing geographic material with users using intelligent online maps. In this scenario, *GeoMessage* Simulator [19] receives XML files as input and generates, in different frequencies, User Datagram Protocol (UDP) broadcast datagrams as an output for ArcGIS's applications and system components. *GeoMessage* is also cross-platform and contains 1209 Qt/C++ code lines using 20 different Qt framework APIs, covering several features, such as Qt event system, strings, file handling, widgets, and so forth. Finally, *GeoMessage* uses two classes, `QMutex` and `QMutexLocker`, related to Qt Threading module (*i.e.*, classes for concurrent programs). Such classes were used in the application to lock/unlock mutexes and, most importantly, ESBMC++ is able to properly verify those structures; however, QtOM does not provide full support for the Qt Threading module yet.

ESBMC++ with QtOM was applied to verify *Locomaps* and *GeoMessage*, while checking the following properties: array-bounds violations, under and overflow arithmetic, division by zero, pointer safety, and other specific properties defined in QtOM (see Section 4). Furthermore, ESBMC++ was able to fully analyse the source code from each application, using five different modules of QtOM for *Locomaps* and twenty modules for *GeoMessage* (*i.e.*, each one corresponding to each API used in the application). The verification process of both applications was totally automatic and the proposed methodology took approximately 6.7 seconds to generate 32 VCs, for *Locomaps*, and 16 seconds to generate 6421 VCs, for *GeoMessage*. Additionally, ESBMC++ did not report any *false incorrect* result, *i.e.*, no error was reported for a program that fulfills the specification; ESBMC++ was also able to find similar bugs in both applications, which were confirmed by the developers (and are explained below).

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

Figure 16. Code fragment from the main file of the *Locomaps* application.

Fig. 16 shows a code fragment from the main file of *Locomaps* application, which uses the `QApplication` class present in the `QtWidgets` module. In that particular case, if the `argv` parameter is not correctly initialized, then the constructor called by object `app` does not execute properly and the application crashes (see line 2, in Fig. 16). According to ISO/IEC 9899 : 1999 [49], `argc` must be nonnegative and if the value of `argc` is greater than zero, the first string pointed by `argv` represents the program name and the remaining ones represent program parameters. In addition, regarding Qt documentation [1], `argv` must point at least to one valid string and the data from `argc` and `argv` must stay valid throughout the entire lifespan of the `QApplication` object (*i.e.*, `app`). Most importantly, Qt documentation also states that "`argc` and `argv` might be changed as Qt removes command line arguments that it recognizes" [1]. Thus, in order to verify this property, QtOM includes two assertions regarding the (input) parameters (see lines 4 and 5, in Fig. 17), by evaluating them as preconditions. A similar bug was also found in the *GeoMessage* application. One possible way to fix such a bug is to always check, with conditional statements, whether `argv` and `argc` are valid arguments, before applying them to an operation.

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

Figure 17. Operational model for the `QApplication()` constructor.

7. CONCLUSION AND FUTURE WORK

This paper proposes an approach to verify C++/Qt programs, using an operational model named as QtOM, which includes pre and postconditions, simulation features (*e.g.*, how element values are manipulated and stored), and also how those are used, in order to verify Qt-based applications. An implementation for the operational model, used to verify sequential and associative containers, was also described, in detail. Additionally, a Qt-based touchscreen application for browsing maps, satellite, and terrain data [18] and another one to generate UDP broadcast datagrams, based on XML files [19], were verified using ESBMC++ with QtOM, which proved the potential of the proposed approach for checking real-world Qt-based applications.

The main contribution of this work is the support for sequential and associative containers, in the Qt framework through QtOM. The performed experiments involved Qt/C++ programs, with many features offered by the Qt container classes. In fact, verification results showed the efficiency and the effectiveness of SMT-based BMC approach with ESBMC++, regarding the verification of Qt programs, and presented, for the developed benchmarks, a success rate of approximately 89.3%, in 1760 seconds.

In addition, the performance of Z3, Boolector, and Yices solvers were also evaluated, given that they were applied in the verification process of Qt programs with ESBMC++ and QtOM. Regarding that, Boolector presented the highest coverage with lowest verification time.

Another fundamental contribution is the integration of QtOM into the verification process of two different model checkers, known as LLBMC and DiVinE, which demonstrates QtOM's flexibility. Such alternatives also show prominent results, given that LLBMC detected 95% (in 2513.5 seconds) and DiVinE found 92% (in 14722.4 seconds) of the existing bugs, overcoming ESBMC++ that detects 89.2% (in 1760 seconds). In contrast, LLBMC reports 18.6% of incorrect results (the highest rate among the model checkers), followed by DiVinE reporting 11.3%, and ESBMC++ with only 10.9%. One may notice that DiVinE is $7x$ slower than the others tools, followed by LLBMC and ESBMC++, respectively. In brief, QtOM can be integrated into a suitable checker and used for checking real C++/Qt implementations, for specific scenarios and applications.

As future work, the developed QtOM will be extended, in order to fully support the verification of multi-threaded Qt software. In addition, more classes and libraries will be integrated, with the goal of increasing the coverage of the Qt framework and validate its properties, and performance measuring tools will be included, in order to check if specific routines comply with time restrictions.

REFERENCES

[1] The Qt Company Ltd. The Qt Framework. http://www.qt.io/qt-framework/ 2017. [Online; accessed January-2017].

[2] LG Electronics. Open webOS. http://www.openwebosproject.org 2016. [Online; accessed January-2017].

[3] Sky UK. Sky Q. The next generation box. http://www.sky.com/shop/tv/sky-q/ 2017. [Online; accessed January-2017].

[4] Advanced Micro Devices, Inc. Radeon Software Crimson ReLive Edition. http://www.amd.com/en-gb/innovations/software-technologies/enhanced-media/radeon-software 2017. [Online; accessed January-2017].

[5] Slobodová, Anna. *Challenges for Formal Verification in Industrial Setting*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2007; 1–22, doi:10.1007/978-3-540-70952-7_1.

[6] Fix, Limor. *Fifteen Years of Formal Property Verification in Intel*, chap. 8. Springer Berlin Heidelberg: Berlin, Heidelberg, 2008; 139–144, doi:10.1007/978-3-540-69850-0_8.

[7] Berard B, Bidoit M, Finkel A, Laroussinie F, Petit A, Petrucci L, Schnoebelen P. *Systems and Software Verification: Model-Checking Techniques and Tools*. 1st edn., Springer Publishing Company, Incorporated, 2010.

[8] Slobodová A, Davis J, Swords S, Hunt W. A flexible formal verification framework for industrial scale validation. *Formal Methods and Models for Codesign (MEMOCODE), 2011* 9_{th} *IEEE/ACM International Conference on*, 2011; 89–97, doi:10.1109/MEMCOD.2011.5970515.

[9] Clarke EM Jr, Grumberg O, Peled DA. *Model Checking*. MIT Press: Cambridge, MA, USA, 1999.

[10] Mehlitz, Peter and Rungta, Neha and Visser, Willem. A Hands-on Java PathFinder Tutorial. *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, IEEE Press: Piscataway, NJ, USA, 2013; 1493–1495. URL http://dl.acm.org/citation.cfm?id=2486788.2487052.

[11] Heila van der Merwe and Brink van der Merwe and Willem Visser. Execution and Property Specifications for JPF-Android. *ACM SIGSOFT Software Engineering Notes* 2014; **39**(1):1–5, doi:10.1145/2557833.2560576. URL http://doi.acm.org/10.1145/2557833.2560576.

[12] Heila van der Merwe and Oksana Tkachuk and Brink van der Merwe and Willem Visser. Generation of Library Models for Verification of Android Applications. *ACM SIGSOFT Software Engineering Notes* 2015; **40**(1):1–5, doi:10.1145/2693208.2693247. URL http://doi.acm.org/10.1145/2693208.2693247.

[13] Ramalho, Mikhail and Freitas, Mauro and Sousa, Felipe and Marques, Hendrio and Cordeiro, Lucas and Fischer, Bernd. SMT-Based Bounded Model Checking of C++ Programs. *Proceedings of the 20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems*, ECBS'13, IEEE Computer Society: Washington, DC, USA, 2013; 147–156, doi:10.1109/ECBS.2013.15. URL http://dx.doi.org/10.1109/ECBS.2013.15.

[14] Monteiro, F R and Cordeiro, L C and de Lima Filho, E B. Bounded Model Checking of C++ Programs Based on the Qt Framework. 4^{th} *Global Conference on Consumer Electronics*, IEEE, 2015.

[15] Garcia, Mário and Monteiro, Felipe and Cordeiro, Lucas and de Lima Filho, Eddie. *$ESBMC^{QtOM}$: A Bounded Model Checking Tool to Verify Qt Applications*, chap. 10. Springer International Publishing: Cham, 2016; 97–103, doi:10.1007/978-3-319-32582-8_6. URL http://dx.doi.org/10.1007/978-3-319-32582-8_6.

[16] Jiří Barnat and Luboš Brim and Vojtěch Havel and Jan Havlíček and Jan Kriho and Milan Lenčo and Petr Ročkai and Vladimír Štill and Jiší Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. *Computer Aided Verification (CAV 2013)*, *LNCS*, vol. 8044, Springer, 2013; 863–868.

[17] Stephan Falke and Florian Merz and Carsten Sinz. The bounded model checker LLBMC. *ASE*, 2013; 706–709, doi:10.1109/ASE.2013.6693138. URL http://dx.doi.org/10.1109/ASE.2013.6693138.

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

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

[20] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *TACAS*, *LNCS*, vol. 4963, 2008; 337–340, doi:10.1007/978-3-540-78800-3_24. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[21] Dutertre, Bruno. Yices 2.2. *Computer-Aided Verification (CAV'2014)*, *Lecture Notes in Computer Science*, vol. 8559, Biere, Armin and Bloem, Roderick (ed.), Springer, 2014; 737–744, doi:10.1007/978-3-319-08867-9_49.

[22] Brummayer, R and Biere, A . Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays . *TACAS* , *LNCS* , vol. 5505 , 2009; 174–177 .

[23] Kroening D, Tautschnig M. CBMC - C bounded model checker - (competition contribution). *TACAS*, *LNCS*, vol. 8413, 2014; 389–391, doi:10.1007/978-3-642-54862-8_26. URL http://dx.doi.org/10.1007/978-3-642-54862-8_26.

[24] Cordeiro, L and Fischer, B and Marques-Silva, J. SMT-based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Trans. Software Eng.* 2012; **38**(4):957–974.

[25] Wang W, Barrett C, Wies T. Cascade 2.0. *VMCAI*, *LNCS*, vol. 8318, 2014; 142–160, doi:10.1007/978-3-642-54013-4_9. URL http://dx.doi.org/10.1007/978-3-642-54013-4_9.

[26] Lattner C. *CLang Documentation*. The Clang-LLVM Project 2015. [Online; accessed December-2015].

[27] Dillig, Isil and Dillig, Thomas and Aiken, Alex. Precise Reasoning for Programs Using Containers. *SIGPLAN Not.* Jan 2011; **46**(1):187–200, doi:10.1145/1925844.1926407. URL http://doi.acm.org/10.1145/1925844.1926407.

[28] Blanc N, Groce A, Kroening D. Verifying C++ with STL containers via predicate abstraction. *ASE*, 2007; 521–524, doi:10.1145/1321631.1321724. URL http://doi.acm.org/10.1145/1321631.1321724.

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

[30] Sites RL. Some thoughts on proving clean termination of programs. *Technical Report*, Stanford, CA, USA 1974.

[31] Merz F, Falke S, Sinz C. LLBMC: bounded model checking of C and C++ programs using a compiler IR. *VSTTE*, *LNCS*, vol. 7152, 2012; 146–161, doi:10.1007/978-3-642-27705-4_12. URL http://dx.doi.org/10.1007/978-3-642-27705-4_12.

[32] Cordeiro LC, Fischer B. Verifying multi-threaded software using SMT-based context-bounded model checking. *ICSE*, 2011; 331–340, doi:10.1145/1985793.1985839. URL http://doi.acm.org/10.1145/1985793.1985839.

[33] Bradley AR, Manna Z. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc.: Secaucus, NJ, USA, 2007.

[34] Mccarthy J. Towards a mathematical science of computation. *In IFIP Congress*, North-Holland, 1962; 21–28.

[35] Barrett C, Tinelli C. CVC3. *CAV*, *LNCS*, vol. 4590, 2007; 298–302.

[36] de Moura LM, Bjørner N. Satisfiability modulo theories: An appetizer. *SBMF*, 2009; 23–36.

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

[38] The Qt Company Ltd. Panasonic Avionics Inflight Entertainment. https://www.qt.io/case-panasonic/ 2017. [Online; accessed January-2017].

[39] The Qt Company Ltd. Zühlke Engineering helps enable rapid development of low noise digital laser control. https://www.qt.io/case-zuehlke/ 2017. [Online; accessed January-2017].

[40] Research2guidance. Cross-Platform Tool Benchmarking. *Technical Report*, Research2guidance 2014.

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

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

[43] The Qt Company Ltd. Text Edit. http://doc.qt.io/qt-5/qtwidgets-richtext-textedit-example.html 2016. [Online; accessed January-2017].

[44] Deitel P, Deitel H. *C++ How to Program*. Prentice Hall, 2013.

[45] Pereira P, Albuquerque H, da Silva I, Marques H, Monteiro F, Ferreira R, Cordeiro L. Smt-based context-bounded model checking for cuda programs. *Concurrency and Computation: Practice and Experience* 2016; doi:10.1002/cpe.3934. URL http://dx.doi.org/10.1002/cpe.3934, cpe.3934.

[46] Pedro de la Cámara and J Raúl Castro and María-del-Mar Gallardo and Pedro Merino. Verification support for ARINC-653-based avionics software. *Softw. Test., Verif. Reliab.* 2011; **21**(4):267–298, doi:10.1002/stvr.422.

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

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

[49] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C. *Technical Report*, International Organization for Standards 1999.