ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Applications

Mário Garcia, Felipe Monteiro, Lucas Cordeiro and Eddie de Lima Filho

Electronic and Information Research Centre, Federal University of Amazonas, Brazil

Abstract We integrate a simplified model of the Qt framework, named as Qt operational model (QtOM), into the efficient SMT-based context-bounded model checker (ESBMC++), which results in ESBMC^{QtOM}. In particular, ESBMC^{QtOM} is a bounded model checking tool to verify Qt-based applications, which focuses on the verification of code properties, such as invalid memory access and containers usage, through pre- and postconditions, data usage evaluation, and simulation features. Experimental results show that ESBMC^{QtOM} can be effectively and efficiently applied to verify Qt-based consumer electronics applications.

1 Introduction

Currently, in order to be competitive, consumer electronics companies tend to provide devices with reduced prices, which are produced in large scale. As a consequence, profit margins tend to be small, which, in turn, acts as a feedback to the production process. In particular, such products must be as robust and bug-free as possible, given that even medium product-return rates tend to be unacceptable. This way, it is important to adopt reliable verification methods, with the goal of ensuring system correctness and avoiding losses [1].

Model checking [1] is an interesting approach, due to the possibility of automated verification, which makes such a process cheap and simple. Nonetheless, the employed verifier should provide support regarding target language and system properties, which even include linked libraries and development frameworks. For instance, a checker based on satisfiability modulo theories (SMT), such as the efficient SMT-based context-bounded model checker (ESBMC++) [2], can be employed to verify C/C++ code, but it does not support specific frameworks, such as Qt [3]. The same happens to Java PathFinder [4], with respect to multimedia home platform applications [5] and programs developed for the Android operating system [6]. The former are even verified with a specific test suite, which relies on a specialized library; however, such a problem could be overcome through an abstract representation of the associated libraries, known as operational model (OM) [7]. Indeed, it must approximate the behavior of the original modules to provide the same inputs and outputs to the main application.

The present work provides a Qt framework OM, which checks properties related to Qt modules, named as Qt Operational Model (QtOM). QtOM was integrated into ESBMC++, which gave rise to ESBMC^{QtOM}, in order to verify specific properties in Qt/C++ programs. It is worth noticing that the combination between ESBMC++ and OMs has been previously applied to verify

Qt/C++ programs [8]; however, the present work largely extended that, in order to verify specific properties related to Qt structures, via pre and postconditions. Besides, two real-world applications were included in the current benchmark set.

Contributions. The present paper extends a previously published work [8]. Here, implementation and usage aspects are tackled and, in particular, QtOM now includes new features from the Qt Essentials modules [3], in order to verify two Qt-based applications: *Locomaps* [9] and *GeoMessage* [10], which were not part of the previous benchmark set [8]. Besides, sequential and associative Qt containers [2] were also included into QtOM. To the best of our knowledge, there is no other bounded model checker for Qt-based programs, regarding consumer electronics devices. All benchmarks, OMs, tools, and experimental results, associated with the current evaluation, are available on a supplementary web page[1].

2 Qt Operational Model (QtOM)

QtOM strictly provides the same behavior of the Qt framework, while presenting a simplified implementation focused on property verification [8], and can be split into functionality modules [3]. The QtOM Core module [3], as also happens to Qt, contains all non-graphical core classes (including containers) and presents a complete abstraction for the graphical user interface part.

QtCore also comprises container classes, which implement template-based containers for general purpose, similar to what is offered by the standard template libraries (STL). For instance, `QVector<QWidget>` or `QStack<QWidget>` could be used for implementing a dynamic array of QWidgets. The former is similar to the STL counterpart, while the latter provides a last in, first out semantics with new methods, such as `QStack::push()`.

Fig. 1 shows the integration of QtOM into ESBMC++, *i.e.*, ESBMC^{QtOM}, where gray boxes represent the respective OM, the white ones show inputs and outputs, the dotted ones belong to ESBMC++, and elements connected through dotted arrows represent the components used to build QtOM. The first step is the parser, where ESBMC++ translates the input code into an intermediate representation (IR) tree, where each language structure is correctly identified, by means of QtOM. The latter considers each library and its associated classes, including attributes, method signatures, and function prototypes, through assertions, as shown in Fig. 2. Indeed, assertions are of paramount importance, given that they ultimately allow formal property verification.

Hence, QtOM aids the parser process to build a C++ IR with all necessary assertions to verify Qt-specific properties. After that, the remaining verification flow is normally carried out, as described by Cordeiro *et al.* [11].

It is clear that the usefulness of the proposed methodology relies on the fact that QtOM correctly represents the original Qt libraries. In that sense, all developed QtOM modules were manually verified and exhaustively compared with the original ones, in order to guarantee the same behavior. Besides, although QtOM is a new implementation, it consists in constructing a simplified model of the related libraries, using the same language and through the original code and documentation, which tends to reduce the resulting number of errors.

[1] http://esbmc.org/qtom/

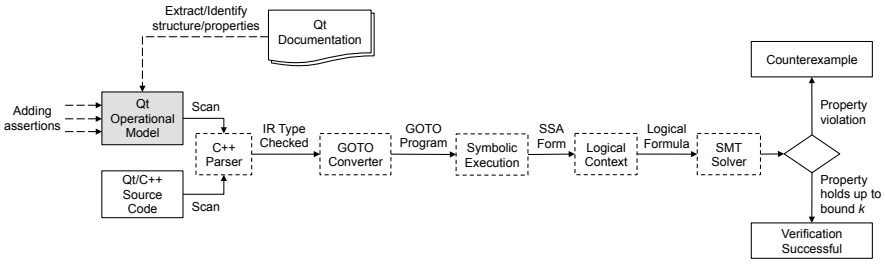

Figure 1: Connecting QtOM to ESBMC++ architecture.

Even so, one may also argue that conformance testing regarding OMs [12] would be a better approach, which is true; however, that option is not available in the present case, albeit it is an interesting possibility for future work.

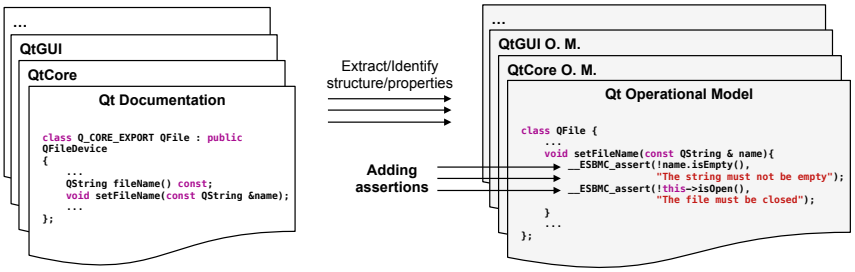

Figure 2: QtOM development process.

3 QtOM Features

Through the integration of QtOM into ESBMC++, ESBMC^{QtOM} is able to properly identify Qt/C++ programs and verify all default properties that it can handle, such as under- and overflow arithmetic, pointer safety, memory leaks, array bounds, and atomicity [2]. Additionally, in order to ensure the correct usage of the Qt methods, pre- and postconditions check the following properties:

- **Invalid memory access.** QtOM assertions ensure that only valid memory addresses are accessed, through operations with arrays, objects, and pointers.
- **Time-period values.** Some Qt features, such as those offered by the `QTime` class, need time-period specifications to be properly executed. This way, QtOM ensures that only valid time parameters are considered.
- **Access to missing files.** The Qt framework provides a set of libraries to handle files (*e.g.*, `QIODevice` and `QFile`). As a consequence, QtOM checks the access and manipulation of all handled files, in a given program.

– **Null pointers.** QtOM also covers pointer manipulation, by adding assertions to ensure that `NULL` pointers are not used in invalid operations.
– **String manipulation.** Unicode character string representations and a set of methods to handle them are provided by the `QString` class. As such structures are widely used by several Qt classes and Qt-based applications, QtOM checks pre and postconditions, for each method from that library, with the goal of ensuring correct string manipulation.
– **Container usage.** The `QtCore` module provides a set of template-based container classes to create collections and provide uniform data management. Due to that, QtOM ensures the correct usage of such structures, as well as their manipulation through specialized methods.

4 QtOM Usage

In order to verify C++ programs based on the Qt framework, users must call the ESBMC++ v1.25.4 command-line version, using

`esbmc <file>.cpp --unwind <k> -I <path-to-QtOM> -I <path-to-C++-OM>`,

where *<file>.cpp* is the Qt/C++ code to be verified, $< k >$ is the maximum loop unrolling, and *<path-to-QtOM>* and *<path-to-C++-OM>* are the locations of the QtOM files and the C++ OM [2], respectively. Thenceforth, the verification process is completely automatic, *i.e.*, if no bug is found, up to a k-depth unwinding, then ESBMC^{QtOM} reports *VERIFICATION SUCCESSFUL*; otherwise, it reports *VERIFICATION FAILED*, along with a counterexample, which contains all necessary information for detecting and reproducing the respective error.

5 Verifying Qt Applications with ESBMC^{QtOM}

5.1 Locomaps Application

ESBMC^{QtOM} was applied to verify a Qt sample application called *Locomaps* [9], which demonstrates satellite, terrain, street maps, tiled map service planning, and Qt Geo GPS Integration, among other features. By means of a unique 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`.

5.2 GeoMessage Application

Another verification was performed on a real-world Qt application called *GeoMessage* simulator, which provides messaging for applications and system components, in the ArcGIS platform [10]. It receives XML files as input and generates, in different frequencies, User Datagram Protocol (UDP) broadcast datagrams, as an output to ArcGIS's applications and system components.

GeoMessage is also cross-platform and contains 1209 Qt/C++ code lines, using 20 different Qt APIs, which cover several features, such as events, file

handling, and widgets. It is worth noticing that *GeoMessage* uses `QMutex` and `QMutexLocker`, which are related to the Qt Threading module (classes for concurrent programs). Such classes were used to lock/unlock mutexes, in *GeoMessage*, and, most importantly, ESBMC^{QtOM} is able to properly verify their structures; however, it does not provide full support to the Qt Threading module yet.

5.3 Verification Results

During the verification of *Locomaps* and *GeoMessage*, the following properties were checked: array-bound violations, under- and overflow arithmetic, division by zero, pointer safety, and other specific properties defined in QtOM (cf. Section 3). Furthermore, ESBMC^{QtOM} was able to fully identify the verified source code, using five different QtOM modules for *Locomaps* and twenty for *GeoMessage*, *i.e.*, one for each original counterpart. The verification process was totally automatic and took approximately 6.7 seconds, for generating 32 verification conditions (VCs) for *Locomaps*, and 16 seconds, regarding 6421 VCs for *GeoMessage*, on a standard PC desktop. Additionally, ESBMC^{QtOM} was able to find similar bugs in both applications, which were confirmed by the respective developers.

Fig. 3 shows a code fragment from *Locomaps*, 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. 3). In order to verify this property, ESBMC^{QtOM} checks two assertions regarding (input) parameters, as can be seen in Fig. 4 (see lines 4 and 5), while evaluating them as preconditions. A similar problem was also found in the *GeoMessage* application. One way to fix such a bug is to check, with conditional statements, whether *argv* and *argc* are valid arguments, before using them in an operation.

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

Figure 3: Code fragment from the main file of the *Locomaps* benchmark.

6 Conclusions

ESBMC^{QtOM} was presented as an SMT-based BMC tool, which employs an operational model (QtOM) to verify Qt-based applications. In particular, QtOM comprises a simple representation of Qt, including several pre- and postconditions, data storage evaluation (*e.g.*, container checks), and simulation features (*e.g.*, string and file manipulation), which are used to check code properties, such as invalid memory access, time-period values, and container usage.

```
class QApplication {
  ...
  QApplication( int & argc, char ** argv ){
  __ESBMC_assert(argc > 0, ''Invalid parameter'');
  __ESBMC_assert(argv != NULL, ''Invalid pointer'');
  this->str = argv;
  this->_size = strlen(*argv);
  ...
 }
 ...
};
```

Figure 4: Operational model for the *QApplication*() constructor.

Additionally, a Qt touch screen program for browsing maps, satellite, and terrain data [9] and another application that provides messaging for the ArcGIS platform [10] were successfully verified, in the context of consumer electronics devices. For the best of our knowledge, there is no other approach, employing BMC, that is able to verify Qt-based applications. For future work, the developed QtOM will be extended (support to more modules), with the goal of increasing the Qt framework coverage. Besides, conformance testing procedures will be developed for validating QtOM, which could also be applied to Qt modules.

References

1. B. Berard, M. Bidoit, A. Finkel: Systems and Software Verification: Model-Checking Techniques and Tool. Springer Publishing, 2010.
2. M. Ramalho *et al.* SMT-Based Bounded Model Checking of C++ Programs. In: ECBS, pp. 147–156, 2013.
3. The Qt Framework. http://www.qt.io/qt-framework/ April, 2015.
4. P. Mehlitz, N. Rungta, W. Visser: A Hands-on Java Pathfinder Tutorial. In: ICSE, pp. 1493–1495, 2013.
5. J. Piesing: The DVB Multimedia Home Platform (MHP) and Related Specifications. Proceedings of the IEEE **94**(1), pp. 237–247, 2006.
6. H. van der Merwe *et al.* Execution and Property Specifications for JPF-Android. ACM SIGSOFT Software Engineering Notes **39**(1), pp. 1–5, 2014.
7. H. van der Merwe *et al.* Generation of Library Models for Verification of Android Applications. ACM SIGSOFT Software Engineering Notes **40**(1), pp. 1–5, 2015.
8. F. Monteiro, L. Cordeiro, E. de Lima Filho: Bounded Model Checking of C++ Programs Based on the Qt Framework. In: GCCE, pp. 179–180, 2015.
9. Spatial Minds and CyberData Corporation: Locomaps. https://github.com/craig-miller/locomaps [accessed 10-September-2015].
10. Environmental Systems Research Institute: GeoMessage Simulator. https://github.com/Esri/geomessage-simulator-qt [accessed 15-September-2015].
11. L. Cordeiro, B. Fischer, J. Marques-Silva: SMT-based bounded model checking for embedded ANSI-C software. IEEE TSE **38**(4), pp. 957–974, 2012.
12. P. de la Cámara, J. Castro, M. Gallardo, P. Merino: Verification support for ARINC-653-based avionics software. JSTVR **21**(4), pp. 267–298, 2011.
13. J. Soulié: C++ Language Tutorial. cplusplus.com. [accessed December-2015].