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 propose a simplified version of the Qt framework, called as Qt Operational Model, which is integrated into the Efficient SMT-based Context-Bounded Model Checking tool, for verifying actual Qt-based applications. Experimental results show that our proposed approach can be effectively and efficiently applied to verify Qt-based real-world applications from consumer electronics.

1 Introduction

Consumer electronics companies 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 [1]. Among such alternatives, one of the most effective and less expensive ways is the model checking approach [1]; 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 [2], but it does not support (full) verification of Java applications that rely on the Android operating system [3]. Indeed, it is true unless an abstract representation of the associated libraries, called operational model (OM), which conservatively approximates their semantics, is available [4].

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, called as Qt Operational Model (QtOM). The developed QtOM is integrated into a bounded model checking (BMC) tool based on satisfiability modulo theories (SMT) solvers, known as the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) [5], to verify specific properties in Qt/C++ programs. Indeed, the combination between ESBMC++ and OMs has been previously applied to verify C++ programs [5]; however, in the current (proposed) methodology, an OM is used to identify elements of the Qt framework and verify specific properties related to such structures, via pre- and post-conditions.

Contributions. The present paper extends a previous published work [6]. We tackle here implementation and usage aspects, related to a tool that provides support for verifying Qt applications. In particular, we improved the QtOM to include new features from the Qt Essentials modules [7], in order to show the verification of two Qt-based applications: *Locomaps* [8] and *GeoMessage* [9]. Given the current knowledge in software verification, there is no other model checker that applies BMC techniques to verify programs based on the Qt framework, regarding consumer electronics devices.

All benchmarks, OMs, tool, and experimental results associated with the current evaluation are available on a supplementary web page[1].

2 Qt Operational Model (QtOM)

QtOM strictly follows the same structure from the Qt framework, but it presents a simplified structure focusing on properties verification [6]. QtOM is subdivide into modules, which are grouped by functionalities, as shown in Fig. 1. As in the Qt framework, QtOM's libraries rely on the Qt Core module [7], which contains all non-graphical core classes. It also presents a complete abstraction for the Graphical User Interface (GUI) part, by using native APIs, from different platforms, to query metrics and draw elements. Based on the Qt's documentation, we identify the classes structures and its properties as shown in Fig. 2. In particular, we create a simpler representation for the respective structure and check each property through assertions, which are included into the QtOM.

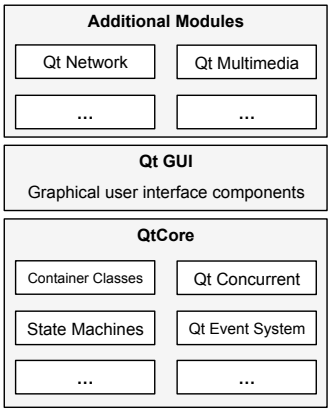

Figure 1: Overview of the QtOM's structure.

Fig. 3 shows the integration of QtOM into ESBMC++ architecture, where the gray box represents the respective OM, the white boxes represent the inputs and outputs, the dotted boxes represent each component from ESBMC++ architecture, and the elements connected by dotted arrows represent the components used to build QtOM. Throughout the verification process with ESBMC++, the first step is the parser, where ESBMC++ translates the input code into an intermediate representation (IR) tree, which encodes all needed information for the verification process. To accomplish this step, ESBMC++ must correctly identify each structure, in the respective program, by means of QtOM, which considers the structure of each library and its associated classes, including attributes, method signatures, and function prototypes; assertions are also integrated into QtOM, in order to ensure that each property is formally checked. Hence, QtOM

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

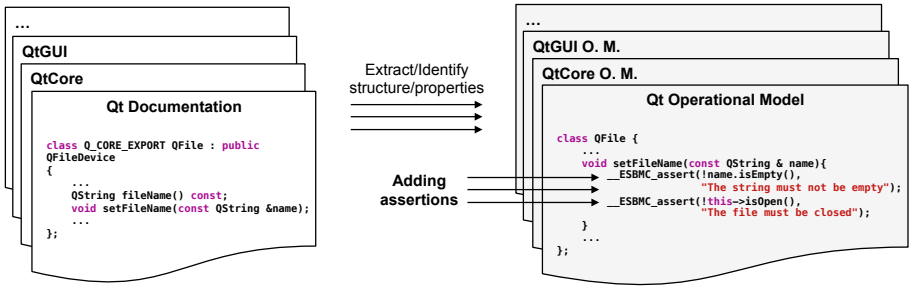

Figure 2: QtOM development process.

aids the parser process to build a C++ IR with all necessary assertions to verify Qt-specific properties; the remaining verification flow is normally carried out, as described by Cordeiro *et al.* [10].

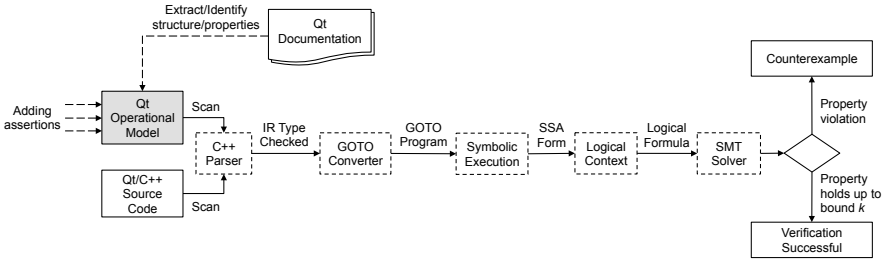

Figure 3: Connecting QtOM to ESBMC++ architecture.

3 QtOM Features

Through the integration of QtOM to ESBMC++, ESBMC^{QtOM} is able to properly identify Qt/C++ programs and verify all default properties, which it can handle, such as under- and overflow arithmetic, pointer safety, memory leaks, array bounds, atomicity and order violations, and so forth [5]. Additionally, in order to ensure the correct usage of the Qt methods, pre- and post-conditions check the following properties:

- **Invalid memory access.** If a certain Qt method attempts to access an unauthorised or nonexistent memory address, then QtOM assertions ensure that only valid memory addresses are accessed by means of operations with arrays, objects, and pointers.
- **Time-period values.** Some Qt functionalities, *e.g.*, those offered by `QTime` class, need time-period specifications to be properly executed. QtOM ensures that only valid time parameters are considered.

– **Access to missing files.** Qt framework provides a set of libraries to handle files, such as `QIODevice` and `QFile`. QtOM checks the access and manipulation of all valid/existing handled files in a program.
– **Null pointers.** If a pointer does not refer to any object or function, it receives a distinguishable value called as `NULL` pointer [11]. QtOM covers pointer operations by adding assertions to ensure that `NULL` pointers are not used in invalid operations.
– **String manipulation.** Unicode character string representation and a set of methods to handle it are provided by `QString` class. Such structures are widely used by several Qt classes and Qt-based applications. To ensure the correct manipulation of strings, QtOM checks the pre- and post-conditions for each method from that library.
– **Containers usage.** The `QtCore` module provides a set of template-based container classes, as an alternative to STL containers [7]. QtOM ensures the correct usage of such structures and the persistence of the stored data.

4 QtOM Usage

To verify C++ programs based on the Qt framework, the user must call the ESBMC++ v1.25.4 command-line version as follows:

`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 and $< k >$ is the maximum loop unrolling, and <*path-to-QtOM*> and <*path-to-C++-OM*> are the locations of the QtOM files and C++ OM [5], respectively. Thenceforth, all the verification process is automatic, *i.e.*, if any bug is found up to k unwindings, then ESBMC^{QtOM} reports *VERIFICATION SUCCESSFUL*; otherwise, if a bug has been found, the tool reports *VERIFICATION FAILED* and the counterexample together with all needed information to detect and reproduce 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* [8], which demonstrates satellite, terrain, street maps, tiled map service planning, and Qt Geo GPS Integration, among other features. By means of an 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, which are `QApplication`, `QCoreApplication`, `QDesktopWidget`, `QtDeclarative`, and `QMainWindow`.

5.2 GeoMessage Application

Another verification was performed on a real-world Qt application called as *GeoMessage* simulator, which provides messaging for applications and system

components, in the ArcGIS platform [9]. It receives XML files as input and generates, in different frequencies, 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. It is worth noticing that *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^{QtOM} is able to properly verify those structures; however, ESBMC^{QtOM} does not provide full support yet to the Qt Threading module.

5.3 Verification Results

To verify the *Locomaps* and *GeoMessage* applications, the following properties were checked: array bounds 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 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 is totally automatic and takes approximately 6.7 seconds to generate 32 verification conditions (VCs) for *Locomaps*, and 16 seconds to generate 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 developers (and are explained below).

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

Figure 4: Code fragment from the main file of the *Locomaps* benchmark.

Fig. 4 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 initialised, then the constructor called by object *app* does not execute properly and the application crashes (see line 2, in Fig. 4). To verify this property, ESBMC^{QtOM} checks two assertions to verify the (input) parameters (see lines 4 and 5, in Fig. 5), 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.

```
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 5: Operational model for the *QApplication*() constructor.

6 Conclusions

This paper presents a tool called ESBMC^{QtOM} to verify C++/Qt programs, using an operational model named as QtOM, which includes pre- and post-conditions, simulation features (*e.g.*, how element values are manipulated and stored), and also how those are used in order to verify Qt-based applications, in consumer electronics devices. Additionally, a Qt touchscreen application for browsing maps, satellite data, and terrain data [8] and another application that provides messaging for ArcGIS platform were successfully verified. For the best of our knowledge, there is no other approach, employing BMC to perform verification on Qt-based applications. As future work, the developed QtOM will be extended, where more classes and libraries will be integrated, with the goal of increasing the Qt framework coverage, in order to verify its properties.

References

1. B. Berard, M. Bidoit, A. Finkel: Systems and Software Verification: Model-Checking Techniques and Tool. Springer Publishing, 2010.
2. P. Mehlitz, N. Rungta, W. Visser: A Hands-on Java Pathfinder Tutorial. In: ICSE, pp. 1493–1495, 2013.
3. H. van der Merwe *et al.* Execution and Property Specifications for JPF-Android. ACM SIGSOFT Software Engineering Notes **39**(1), pp. 1–5, 2014.
4. 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.
5. M. Ramalho *et al.* SMT-Based Bounded Model Checking of C++ Programs. In: ECBS, pp. 147–156, 2013.
6. 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.
7. The Qt Framework. http://www.qt.io/qt-framework/ April, 2015.
8. Locomaps: Spatial Minds and CyberData Corporation. https://github.com/craig-miller/locomaps [accessed 10-September-2015].
9. Environmental Systems Research Institute: GeoMessage Simulator. https://github.com/Esri/geomessage-simulator-qt [accessed 15-September-2015].
10. 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.
11. J. Soulié: C++ Language Tutorial. cplusplus.com. [accessed December-2015].

Appendix - Oral Presentation Plan

The oral presentation plan consists of three parts. First, a brief motivation about general bugs in Qt-based applications and how bounded model checking techniques can help detect such problems. Second, an overview about the ESBMC^{QtOM} implementation and its verification approach. Third, a quick demonstration of ESBMC^{QtOM} command-line version.

1. Introduction and Motivation (20%)

   - Brief introduction about Qt cross-platform framework
   - Common bugs in Qt-based applications
   - Why do we apply bounded model checking techniques for verification of consumer electronics devices?

2. ESBMC^{QtOM} Implementation Aspects (20%)

   - Properties supported by ESBMC^{QtOM}
   - ESBMC^{QtOM} implementation and architecture

3. ESBMC^{QtOM} Demonstration (60%)

   In this part, we will demonstrate the ESBMC^{QtOM} usage, based on our set of benchmarks.

   Video demonstration: http://esbmc.org/qtom/documentation/

The tool and OMs are available for downloading at:

ESBMC^{QtOM}: http://esbmc.org/qtom