23rd International SPIN symposium on Model Checking of Software

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

Mário A. P. Garcia , Felipe R. Monteiro,
Lucas C. Cordeiro, and Eddie B. de Lima Filho

Why should we ensure software reliability?

- Consumer electronic products must be as robust and bug-free as possible, given that even medium product-return rates tend to be unacceptable
  - In 2014, Apple revealed a bug known as *Gotofail*, which was caused by a single misplaced "*goto*" command in the code
  - "*Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS*" – Apple Inc., 2014
  - "*Mozilla browser has around 20,000 open bugs*" – Michail, Amir. ICSE'05.

- It is important to adopt reliable verification methods, with the goal to ensure system correctness

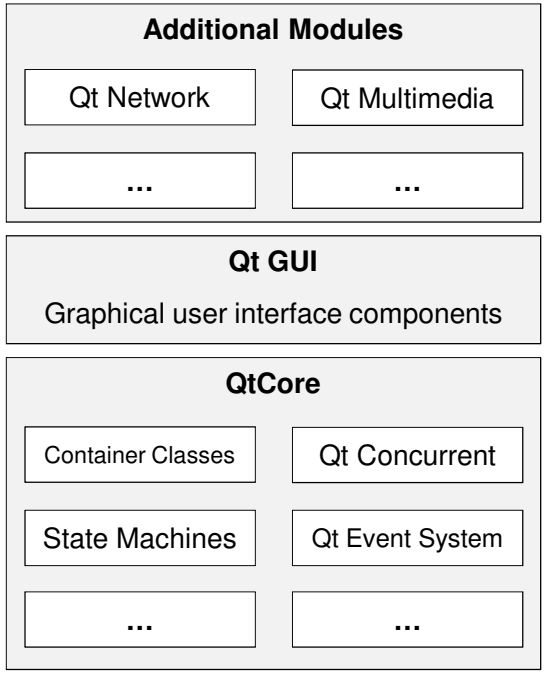
- Model checking is an interesting approach, due to the possibility of automated verification

Model Checkers Limitations

- Verifiers should provide support regarding target language and system properties
  - However, verifiers present limitations to support linked libraries and development frameworks
- Java PathFinder is able to verify Java code, based on byte-code, but it does not support (full) verification of Java applications that rely on the Android operating system
- Efficient SMT-based context-bounded model checker (ESBMC++), can be employed to verify C/C++ code, but it does not support specific frameworks, such as Qt
  - these aspects restrict the range of applications to be verified

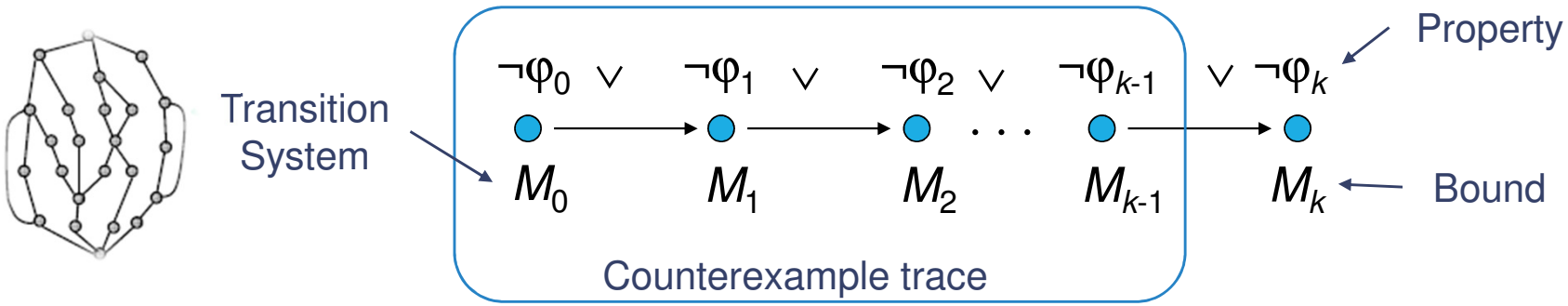
Qt Cross-Platform Framework

- Qt framework provides programs that run on different hardware/software platforms, with as few changes as possible, while maintaining the same power and speed.
  - Samsung, Philips, and Panasonic are some companies, on top 10 fortune 500 list, which apply Qt in their applications development
- Its libraries are organized into modules that rely on two main cores:
  - **QtCore** – contains all non-graphical core classes
  - **QtGUI** – provides a complete abstraction for the Graphical User Interface



Bounded Model Checking (BMC)

- Basic Idea: given a transition system *M*, check negation of a given property *φ* up to given depth *k*



- Translated into a VC *ψ* such that: ***ψ* is satisfiable iff *φ* has counterexample of max. depth *k***
- BMC has been applied successfully to verify (embedded) software since early 2000's

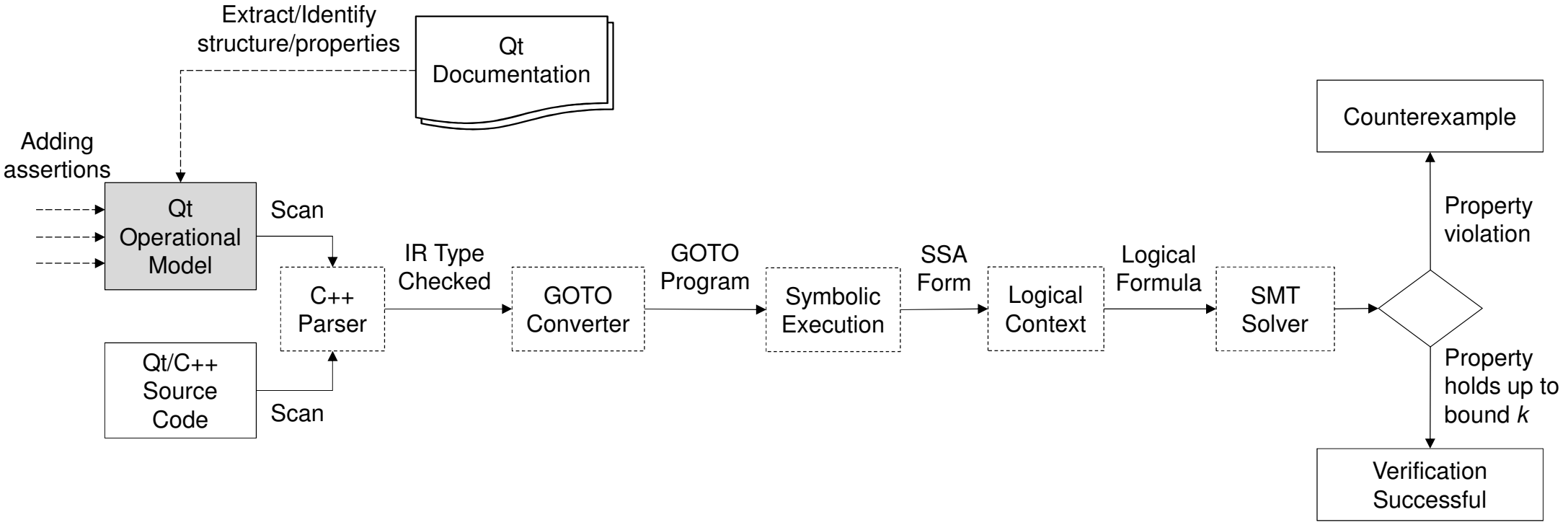
Objectives

Apply bounded model checking to Qt-based applications

- Develop a simplified implementation, which strictly provides the same behaviour of the Qt framework, focused on property verification
- Verify invalid memory access, time-period values, access to missing files, null pointers, string manipulation, container usage, among other properties of Qt applications
- Apply the proposed verification methodology to real world Qt-based applications, which is supported by ESBMC^{QtOM}

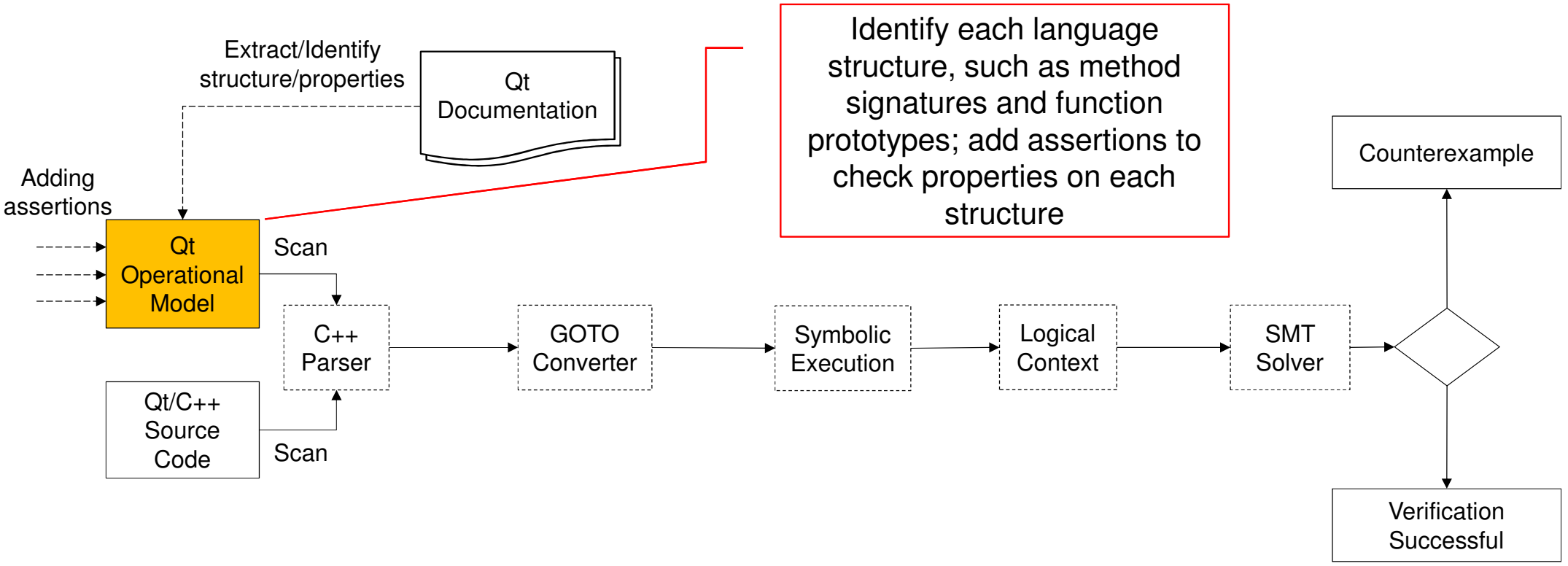
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



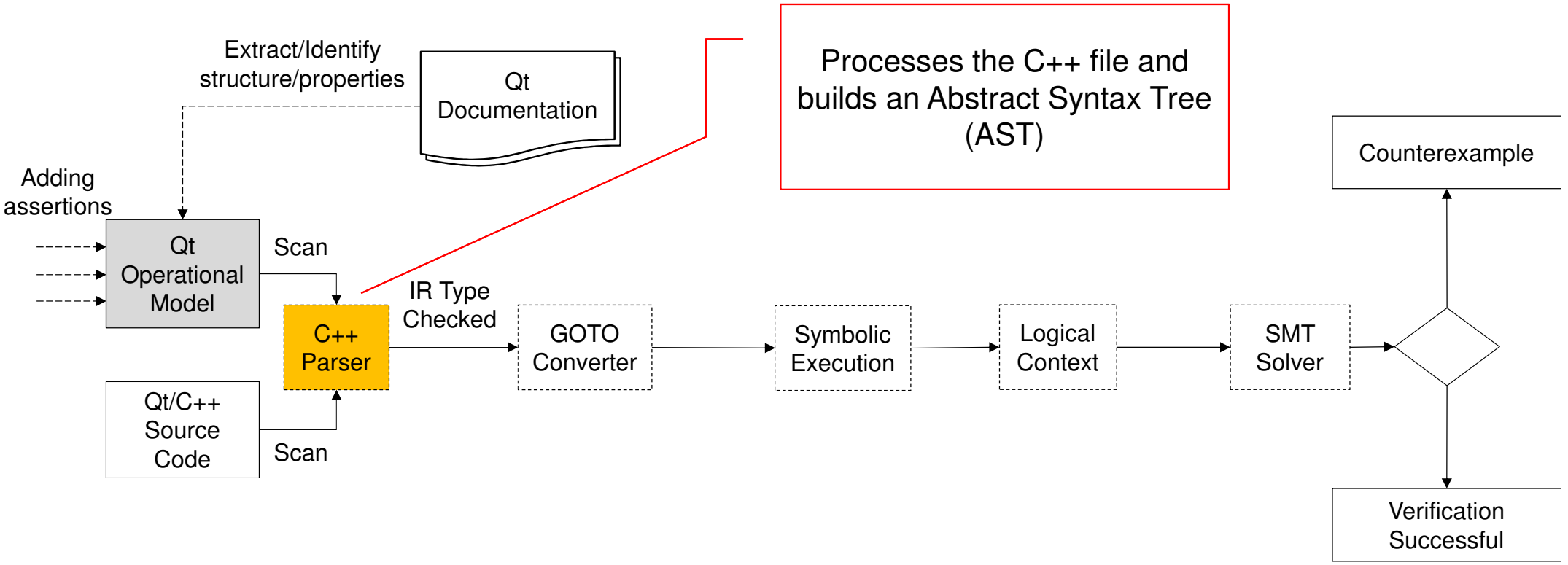
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



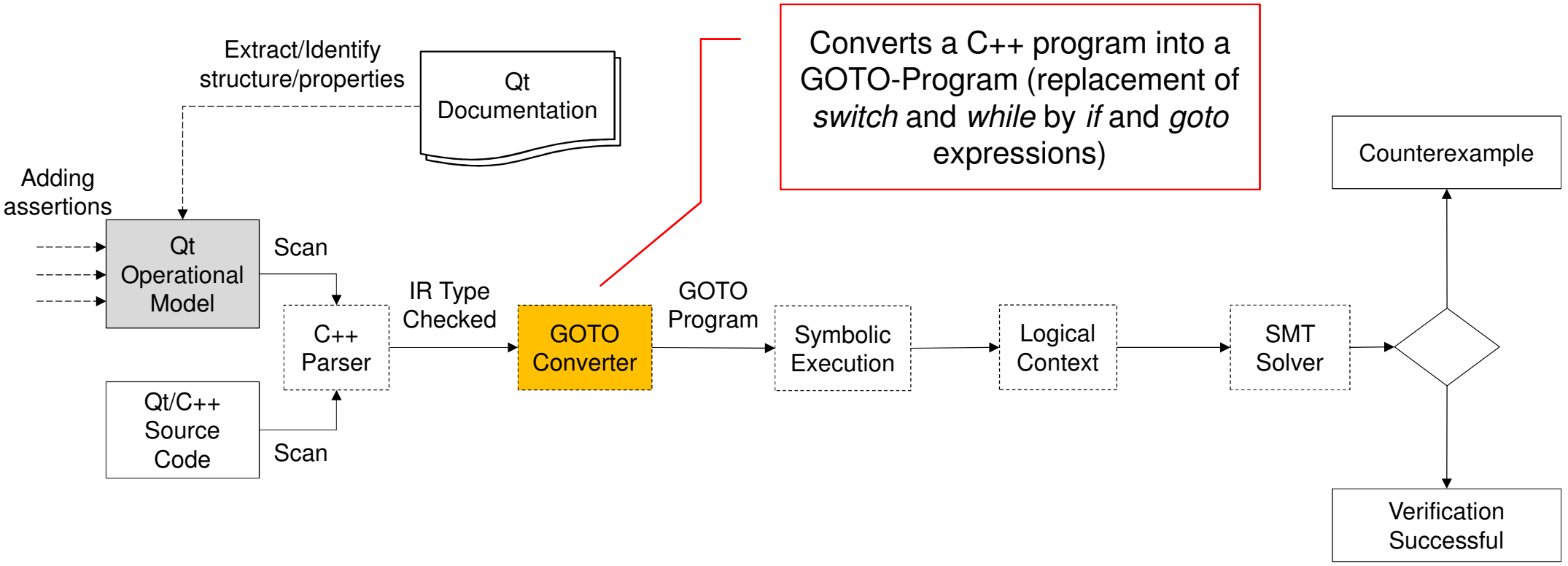
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



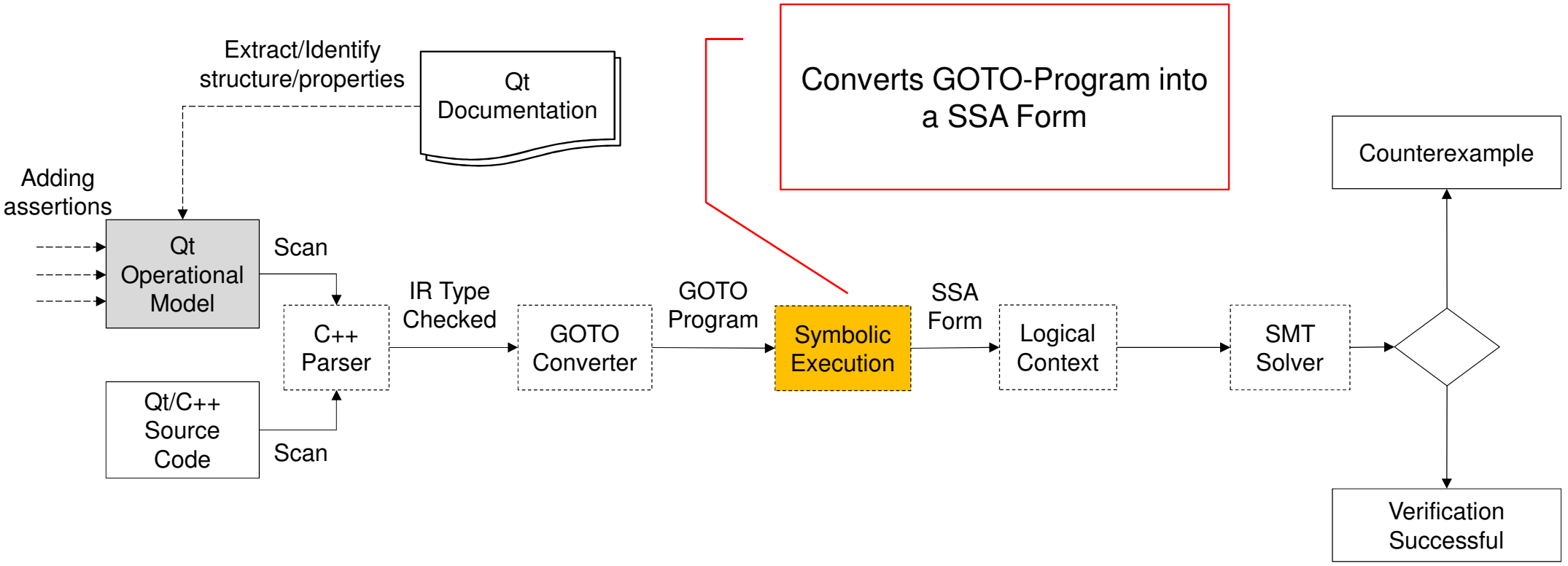
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



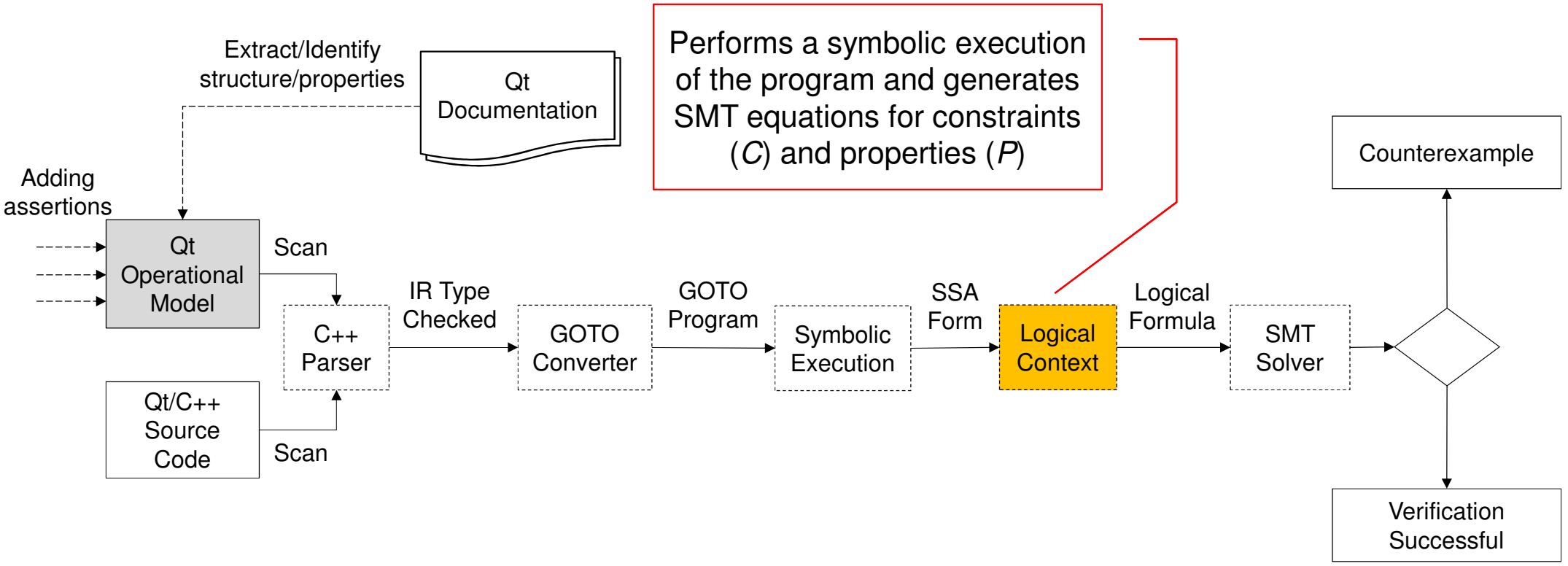
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



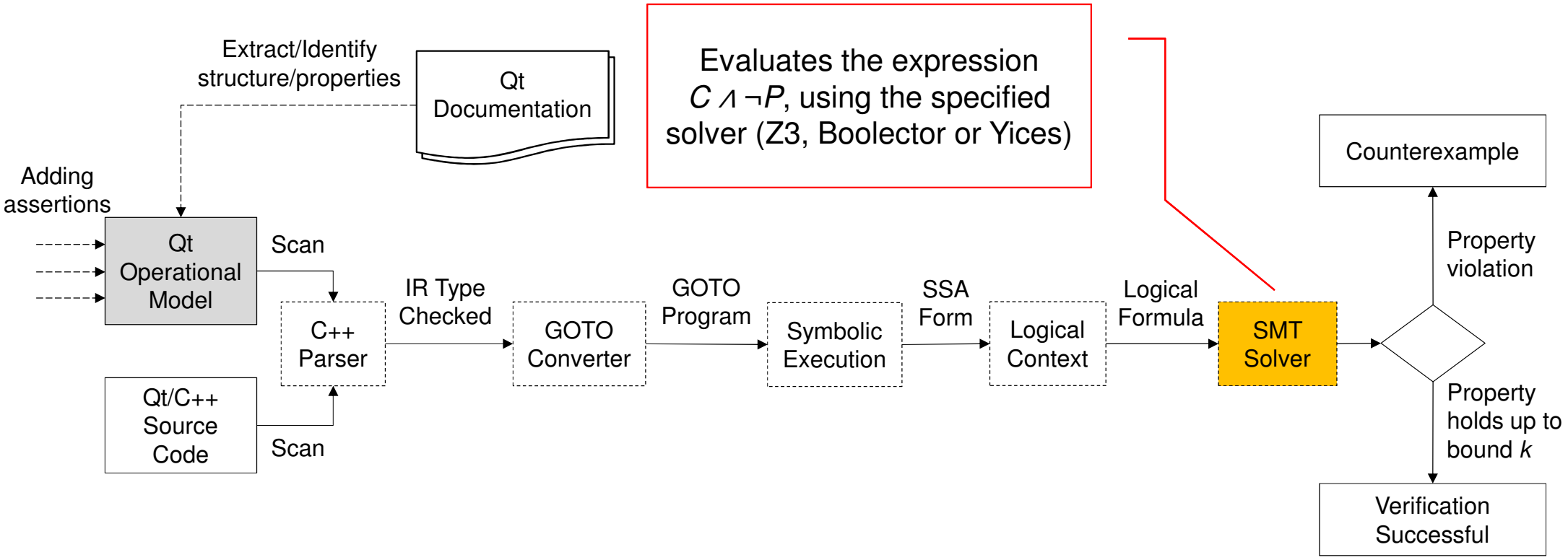
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



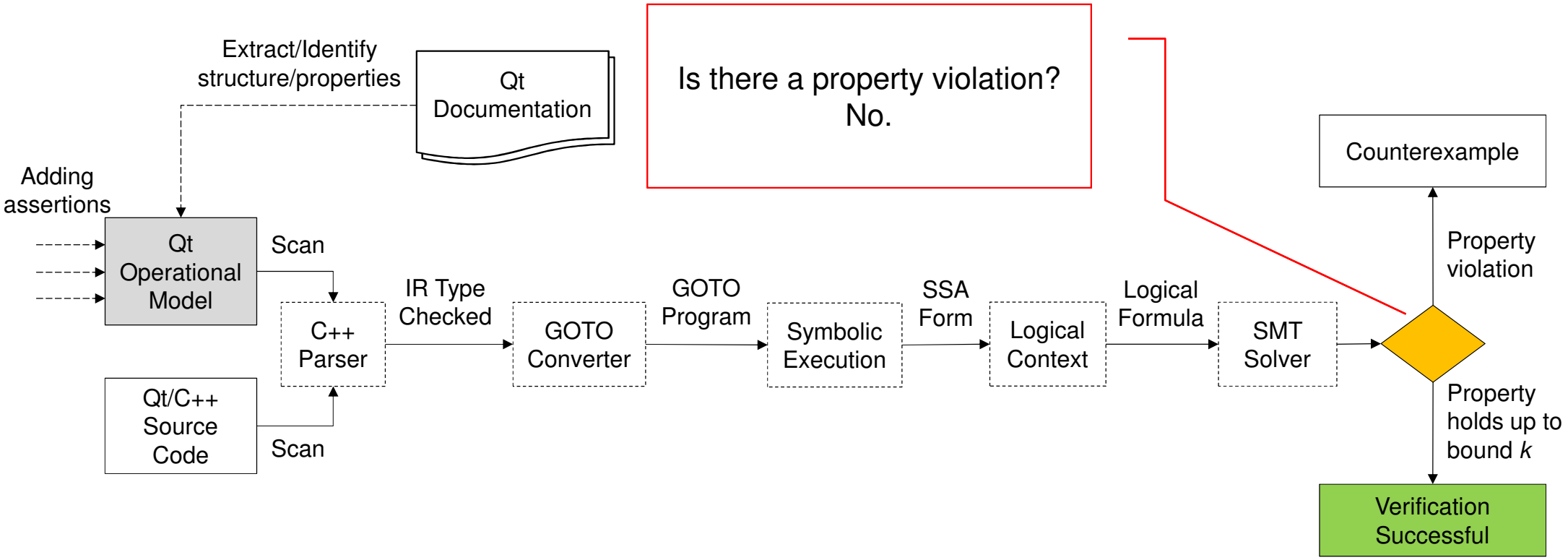
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



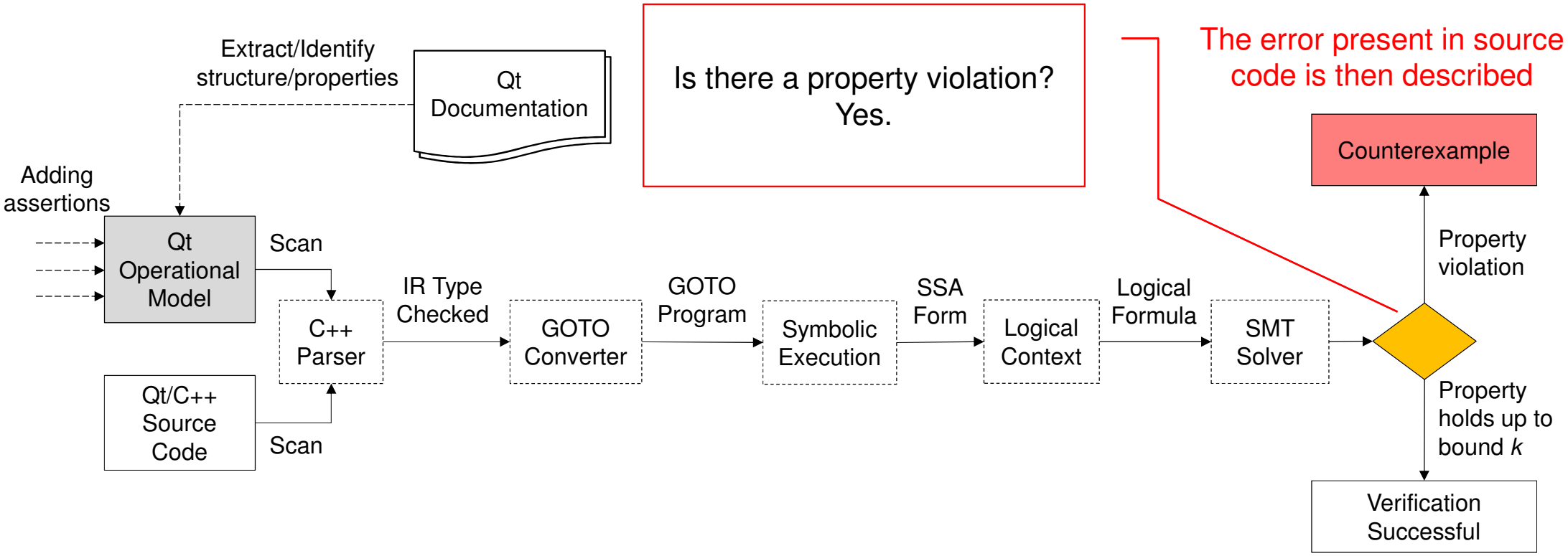
ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



ESBMC^{QtOM} Architecture

- QtOM is an additional extension for the Efficient SMT-based Context-Bounded Model Checker (ESBMC++) to add support for Qt/C++ programs verification.



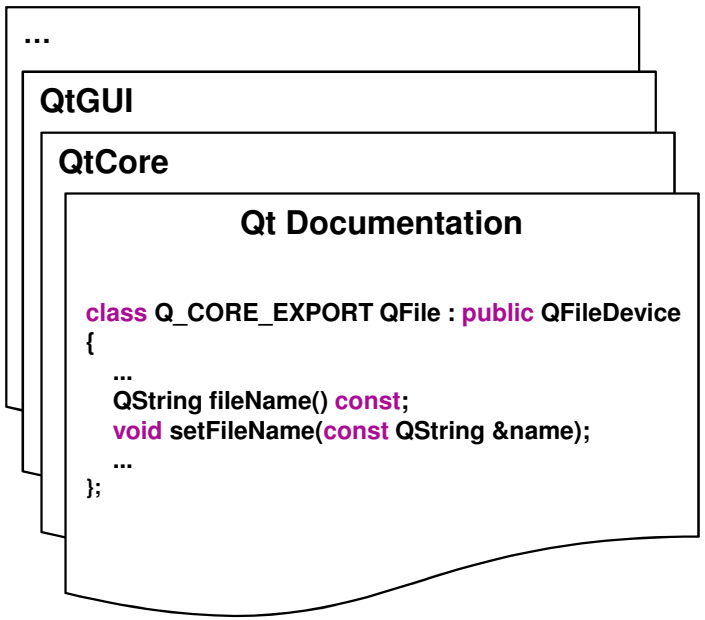
ESBMC^{QtOM} Features

- Through the integration of QtOM into ESBMC++, ESBMC^{QtOM} is able to properly identify Qt/C++ programs and it provides support to the verification of six properties:

1. **Invalid Memory Access:** QtOM assertions ensure that only valid memory addresses are accessed
2. **Time-period values:** QtOM ensures that only valid time parameters are considered
3. **Access to Missing Files:** QtOM checks the access and manipulation of all handled files
4. **Null Pointers:** QtOM covers pointer manipulation, by ensuring that NULL pointers are not used in invalid operations
5. **String Manipulation:** QtOM checks pre- and postconditions to ensure correct string manipulation
6. **Container usage:** QtOM ensures the correct usage of containers, as well as their manipulation through specialized methods

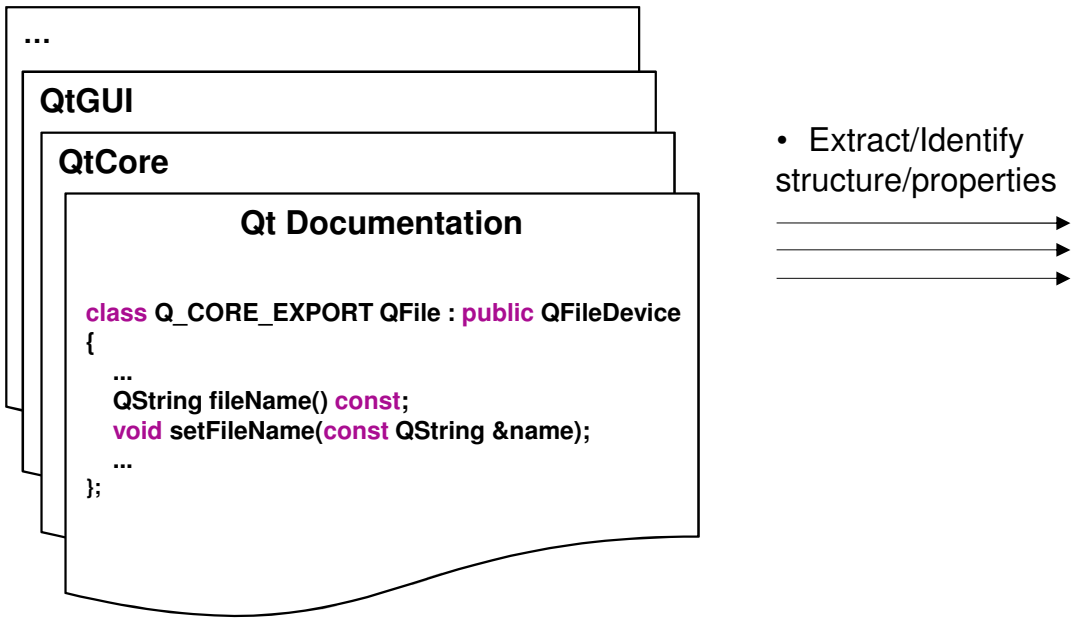
Qt Operational Model (QtOM)

- Development process of the operational model



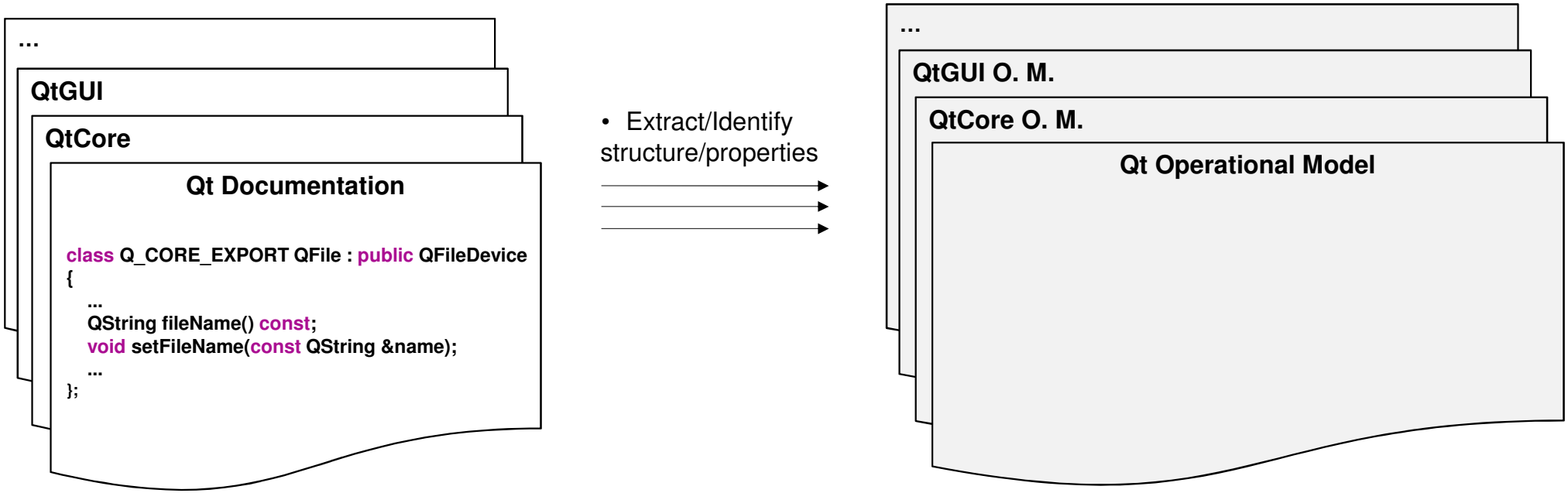
Qt Operational Model (QtOM)

- Development process of the operational model.
  - Identify each property to be verified and transcript them into assertions



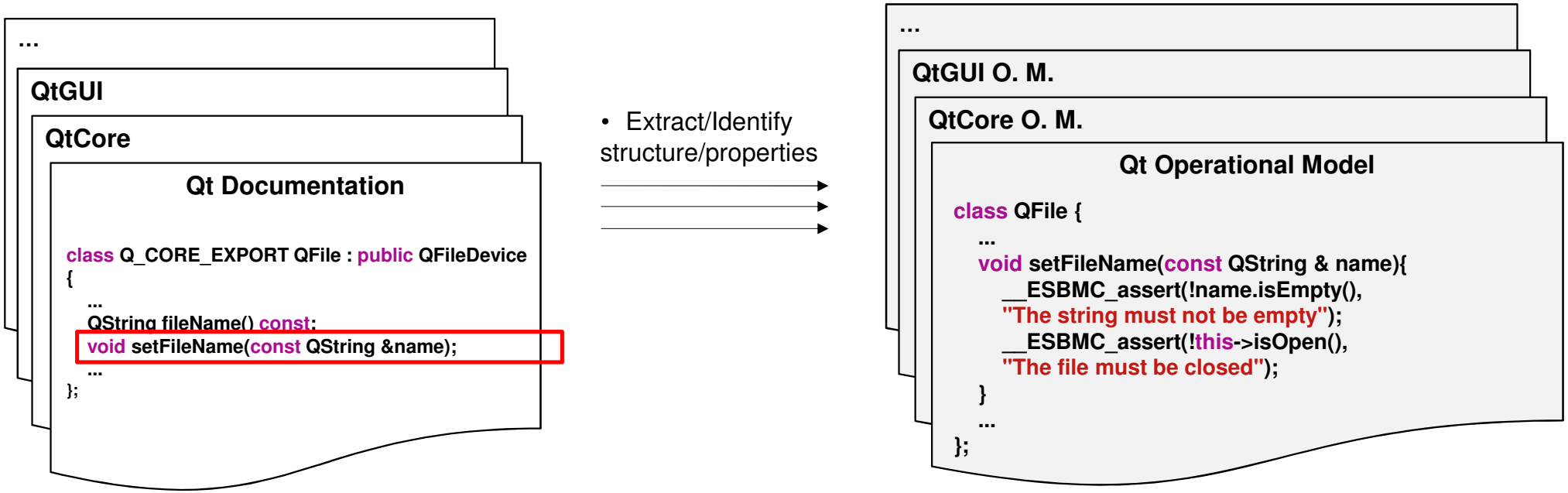
Qt Operational Model (QtOM)

- Development process of the operational model.
  - Qt Operational Model is an abstract representation, which is used to identify elements and verify specific properties related to such structures



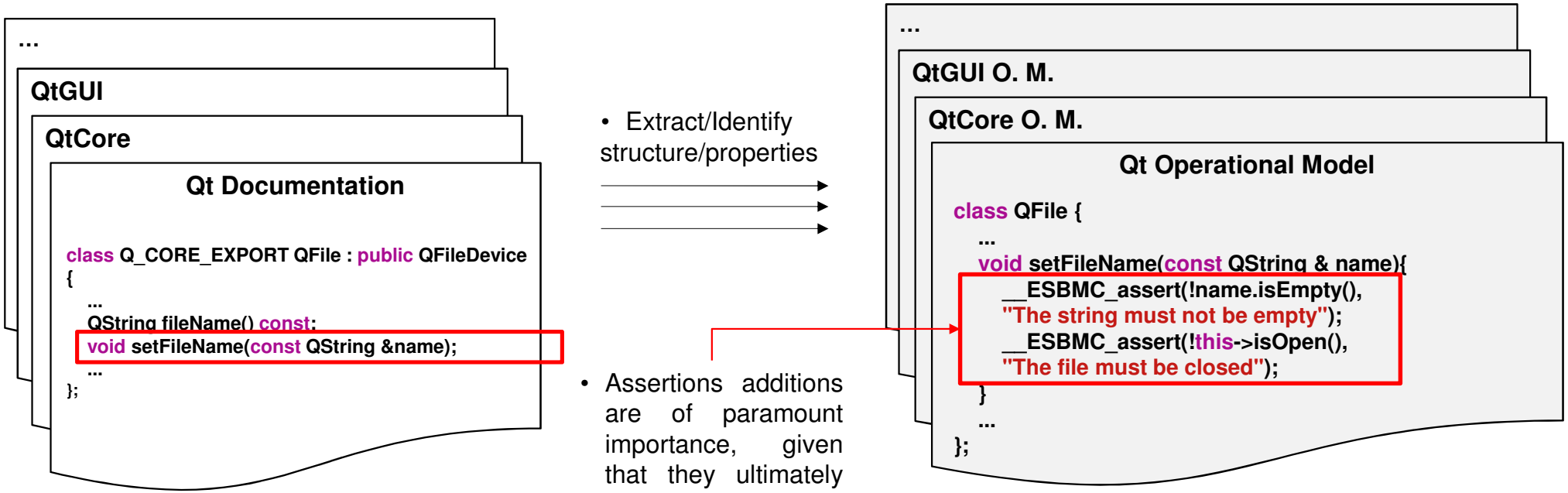
Qt Operational Model (QtOM)

- Development process of the operational model.
  - Qt Operational Model is an abstract representation, which is used to identify elements and verify specific properties related to such structures.



Qt Operational Model (QtOM)

- Development process of the operational model.
  - Qt Operational Model is an abstract representation, which is used to identify elements and verify specific properties related to such structures.



Running Example

- Function from a real-world Qt-based application called GeoMessage
  - Function responsible for loading initial settings of the application from a file

```
QString loadSimulationFile( const QString &fileName )
{
        m_inputFile.setFileName( fileName );

    // Check file for at least one message
    if  ( !doInitialRead() )
    {
      return Qstring( m_inputFile.fileName()
                              + “is an empty message file” );
    }
    else
    {
       return NULL;
    }
}
```

Source Code

Running Example

- Function from a real-world Qt-based application called GeoMessage
  - Function responsible for loading initial settings of the application from a file

```
QString loadSimulationFile( const QString &fileName )
{
        m_inputFile.setFileName( fileName );

    // Check file for at least one message
    if  ( !doInitialRead() )
    {
      return Qstring( m_inputFile.fileName()
                                    + “is an empty message file” );
    }
    else
    {
        return NULL;
    }
}
```

Source Code

- Sets the file name

Running Example

- Operational model of the QFile class
  - Such class contain the representation of the *setFileName* method

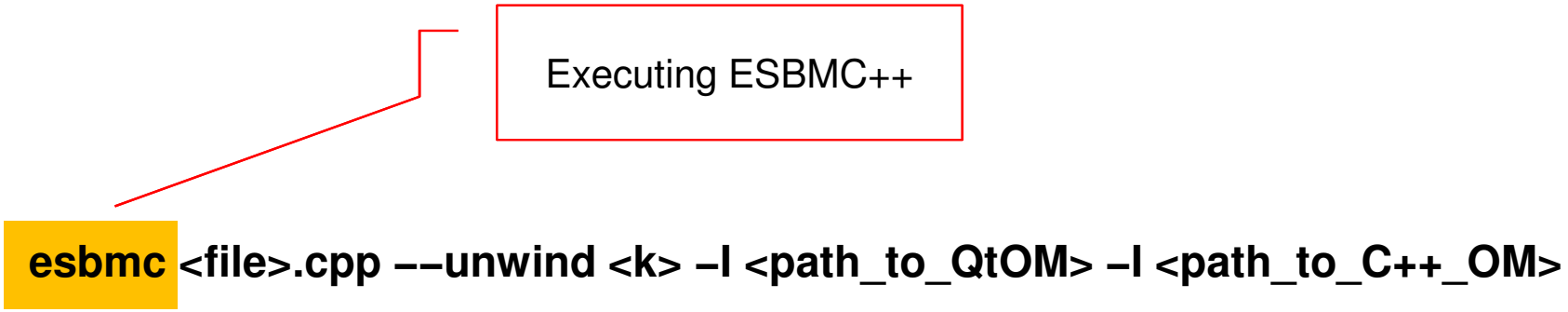
```
class QFile{
        ...
        QFile(const QString& name) { ... }
        ...
        void setFileName( const QString& name){
                __ESBMC_assert(!(name.isEmpty()),
                                "The string must not be empty");
                __ESBMC_assert(!(this->isOpen()),
                                "The file must be closed");
        }
        ...
};
```

Operational Model

- Assertions to handle two properties verification. One checks whether the name is not an empty string and another checks if the file is closed

Running Example

- ESBMC^{QtOM} is invoked as:
  - The verification process is completely automatic



Running Example

- ESBMC^{QtOM} is invoked as:
  - The verification process is completely automatic

Source code to be verified, which in this case is GeoMessage's function

esbmc <file>.cpp ––unwind <k> –I <path_to_QtOM> –I <path_to_C++_OM>

Running Example

- ESBMC^{QtOM} is invoked as:
  - The verification process is completely automatic

Setting up the bound, which in this case we consider as *10*

esbmc <file>.cpp ––unwind <k> –I <path_to_QtOM> –I <path_to_C++_OM>

Running Example

- ESBMC^{QtOM} is invoked as:
  - The verification process is completely automatic

Linking the operational models QtOM and COM for the Qt framework and C++ language, respectively.

esbmc <file>.cpp ––unwind <k> –I <path_to_QtOM> –I <path_to_C++_OM>

Running Example

- Even assuming that a non-empty string is passed to the function, the model checker reports a bug in the source code

```
QString loadSimulationFile( const QString &fileName )
{
        m_inputFile.setFileName( fileName );

    // Check file for at least one message
    if  ( !doInitialRead() )
    {
      return Qstring( m_inputFile.fileName()
                                    + “is an empty message file” );
    }
    else
    {
        return NULL;
    }
}
```

Source Code

There is nothing to ensure that the file was not open previously

Running Example

- After the modification mentioned below, the verification process returns a **successful** result

```
QString loadSimulationFile( const QString &fileName )
{
        if ( m_inputFile.isOpen() )
            m_inputFile.close();
        m_inputFile.setFileName( fileName );

    // Check file for at least one message
    if  ( !doInitialRead() )
    {
       return Qstring( m_inputFile.fileName()
                                    + “is an empty message file” );
    }
    else
    {
        return NULL;
    }
}
```

Source Code

We propose the addition of a conditional structure to ensure the file is closed before setting its name

Qt-based Applications

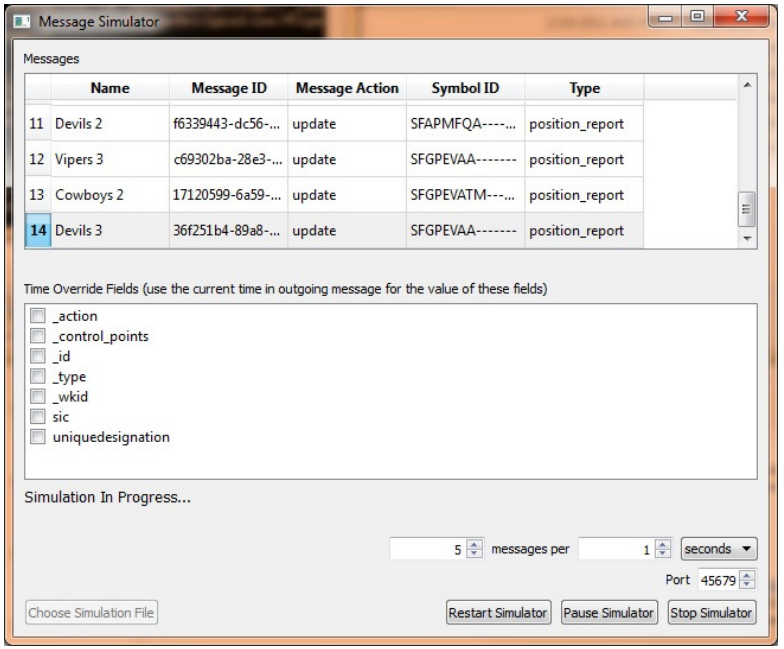
Locomaps Application

- A Qt cross-platform sample application, which demonstrates satellite, terrain, street maps, tiled map service planning, and Qt Geo GPS Integration, among other features.
- Application description:
  - 2 classes
  - 115 Qt/C++ source lines of code
  - 5 different APIs from Qt framework are used
- Experimental Setup
  - ESBMC++ v1.25.4
  - Intel Core i7-2600 computer with 3.40GHz clock and 24GB of RAM

Qt-based Applications

GeoMessage Application

- A real-world Qt application
  - 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.



- Application description:
  - 4 classes
  - 1209 Qt/C++ source lines of code
  - 20 different APIs from Qt framework are used
    - Among them are QMutex and QMutexLocker, which are related to the Qt Threading module

- Experimental Setup
  - ESBMC++ v1.25.4
  - Intel Core i7-2600 computer with 3.40GHz clock and 24 GB of RAM

Experimental Results

Verification Results

- Properties checked:
  - Array-bound violations
  - Under- and overflow arithmetic
  - Division by zero
  - Pointer safety
  - Containers usage
  - String manipulation
  - Access to missing files
- The tool is able to fully identify the verified source code, using 5 different QtOM modules for Locomaps and 20 for GeoMessage.

Experimental Results

- **Verification Results**

- The verification process was automatic and approximately 6.7 seconds for Locomaps and 16 seconds for GeoMessage.
- All the process generated 32 verification conditions(VCs) for Locomaps and 6421 VCs for GeoMessage, on a standard PC desktop.
- ESBMC^{QtOM} is able to find a similar bugs in both applications.

Experimental Results

- 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

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

Source Code

There is nothing to ensure that the parameters are valid

Conclusions

- ESBMC^{QtOM} was presented as an SMT-based BMC tool, which employs an operational model (QtOM) to verify Qt-based applications.
- The performed experiments involved two Qt/C++ applications, which were successfully verified, in the context of consumer electronics devices.
- To the best of our knowledge, there is no other approach, employing BMC, that is able to verify Qt-based applications.

Future Work

- QtOM will be extended to support more modules, with the goal of increasing the Qt framework coverage.
- Conformance testing procedures will be developed for validating QtOM.
- All benchmarks, OMs, tools, and experimental results are available at **http://esbmc.org/qtom**

Demonstration