SMT-Based Bounded Model Checking of C++ Programs

No Author Given

No Institute Given

Abstract. Bounded Model Checking (BMC) of C++ programs presents greater complexity than that of C programs due to the features that the language offers, such as templates, containers, and exception handling. We present ESBMC++, a bounded model checker for C++ programs, which encodes the verification conditions via an operational model using different background theories supported by an SMT Solver. This operational model is an abstract representation of the standard C++ library, which conservatively approximates its semantics. Our experimental results show that our approach can handle a wider range of the C++ constructs than existing C++ model checkers and substantially reduce the verification time.

1 Introduction

Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) solvers has already been successfully applied to verify software and to discover subtle errors in real systems [4]. In an attempt to cope with growing system complexity, SAT solvers are increasingly replaced by Satisfiability Modulo Theories (SMT) solvers to prove the generated verification conditions (VCs) [10–12]. There have also been attempts to extend BMC to the verification of C++ programs [3, 13]. The main challenge here is to handle large programs and support the features that the languages offers, such as templates, containers and exception handling.

C++ is widely used, but systems that make use of the C++ language and its functionalities tend to require a high verification effort since the errors are hard to find. C++ verification involves many more challenges than that of plain ANSI-C since it provides a wider set of features (e.g., object-oriented programming), libraries (e.g., specialized input-output), and functionalities (e.g., template usage) that would require too much effort to develop from scratch. In order to be attractive for mainstream software development, C++ model checkers have to exhibit several crucial characteristics (e.g., speed, accuracy, efficiency, and friendliness).

To tackle this problem, our proposed approach aims to apply Bounded Model Checking (BMC) to C++ programs using an operational model of the C++ libraries. This operational model is an abstract representation of the standard C++ library, which conservatively approximates its semantics. To support this operational mode, we extended our Efficient SMT-Based Bounded Model Checker (ESBMC) tool [12] that builds on the front-end of the C Bounded Model Checker (CBMC) to support C++ features such as inheritance, polymorphism, templates, and exception handling.

In particular, we develop novel approaches to handle excpetions in C++ code that previous approaches are not able to deal with [13, 2]. Additionally, we focus on the

STL sequential containers (which have the most popular data structures in Computing Science) operational model, its preconditions and simulation features (e.g., how we store the elements values of the containers and intern class methods), and how these are used in order to verify real-world C++ programs.

2 Background

ESBMC++ implements BMC for C++ programs using SMT solvers. It can process C/C++ code into equivalent GOTO-programs (i.e., control-flow graphs). The GOTO-programs can then be processed by the symbolic execution engine. ESBMC++ uses CBMC's internal parser, which is based on the OpenC++ [14], to process the C/C++ files and to build an abstract syntax tree (AST). The typechecker of ESBMC++s front end annotates this AST with types and generates a symbol table. ESBMC++s IRep class then converts the annotated AST into an internal, language-independent format used by the remaining phase of the front-end.

ESBMC++ uses two recursive functions that compute the constraints (i.e., assumptions and variable assignments) and properties (i.e., safety conditions and user-defined assertions). In addition, ESBMC++ automatically generates safety conditions that check for arithmetic overflow and underflow, array bounds violations, and NULL-pointer dereferences, in the spirit of Sites clean termination [1]. Both functions accumulate the control flow predicates to each program point and use these predicates to guard both the constraints and the properties so that they properly reflect the programs semantics. ESBMC++s VC generator (VCG) then derives the VCs from these.

3 C++ Operational Model

During the verification process, ESBMC++ has to identify all the specifications and features of the C++ program to generate the AST. The specifications are related to the definitions of the standard C++ libraries such as classes, methods, and types while the features (e.g., inheritance, template, and exception handling) are treated internally in ESBMC++ in different levels (i.e., scan, parser, and type-check). In this sense, we developed a simplified representation of the C++ libraries called C++ Operational Model (COM) to represent the classes, methods, and other features.

The development process of the COM can be divided into two phases: *structural* and *modeling*. In the structural phase, we built a set of classes with a specific hierarchical relationship and the signature of their methods and specific data, which resulted in a simplified structure to represent the set of the standard C++ libraries. From this structure, we modeled the methods of each class and this modeling is focused on the verification of all properties that a specific method includes.

3.1 Structure and Model

The goal here is to build an operational model of the standard C++ library so that ESBMC++ can rely on it to identify the definitions and to verify all the properties related to

these definitions. This operational model is inserted into the verification process at the level of the source code (i.e., both model and program are passed as parameters at the beginning of the verification process). The first step in the construction process of the operational model is the creation of a simplified structure, which is similar to the actual structure. Based on the documentation of the C++ language [16] the set of libraries was split, according to their functionalities, into four subsets called C Libraries, Input/Output Stream, Standard Template Libraries, and General Libraries. Each library presents in a subset is particularly related to the other libraries, not only in its functionality but also in its structure due to the fact that many libraries depend on definitions of others. From this, we performed an analysis in the respective subsets as shown in Figure 1 in such a way that we could identify dependencies between each library and thus develop a simplified structure of each one. In the next sections, we describe the structure of the libraries subsets shown in Figure 1.

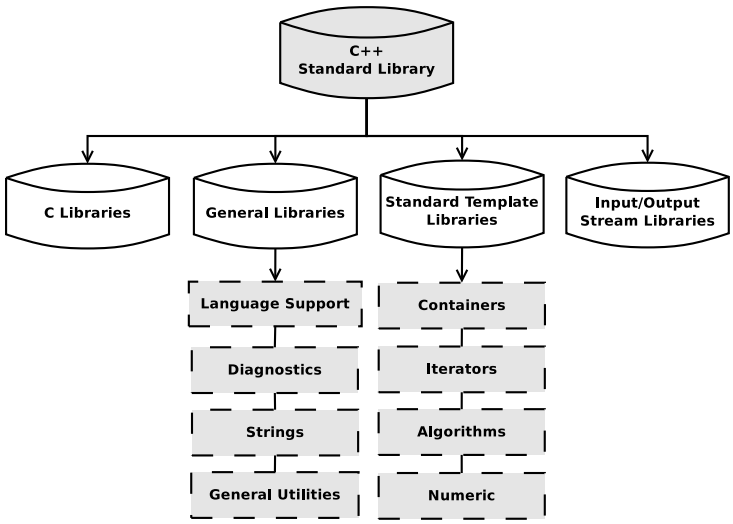

Fig. 1. Representation of the configuration and functional classification of the operational model.

3.2 C Libraries

The standard C++ libraries also include all the ANSI-C libraries. However, for the verification of C++ programs, ESBMC++ follows a different path to parse the C++ programs. For this reason, we also have to build a representation of the ANSI-C libraries in the operational model. We can thus define this libraries' set in a simplified structure, which consists essentially of macro definitions and functions. From this ANSI-C libraries' set, there are libraries that contains only macro definitions (e.g., the *ciso646* library defines a spelling set to the logic operators). To build the structure of this library, we distingue two kinds of spelling: we define the macro set and operators set. For

convention, we assume the macro set M as follows:

$$\{and, and\_eq, bitand, bitor, compl, not, not\_eq, or, or\_eq, xor, xor\_eq\} \subset M \quad (1)$$

The operator set O is defined as follows:

$$\{\ \&\&,\ \&=,\ \&,\ |,\tilde{},\ !,\ ||,\ |=,\ \hat{},\hat{}=\ \} \subset O \quad (2)$$

Therefore, we define the syntax for these definitions sets as:

$$(\alpha \in M \wedge \beta \in O) \Rightarrow (\forall\alpha)\,(\exists\beta)\,(\alpha \equiv \beta) \quad (3)$$

Within the ANSI-C libraries set, we can also represent the structure of others libraries with functions only. This is case of the *csignal* library, which deals with signals that are emitted to a given code. We thus define that the operational model of this library can been represented by signals that raise functions. For a vector v of size N and for a finite signals set S where $\{s_1, \ldots, s_n\} \subset S$ and $n < N$, the function signal records a certain function in one specific signal such that when the raising function is called with the respective signal, the registered function is called. Let f be a function, φ be a signal, and η be a NULL element. We can thus model the signal of the *csginal* library as:

$$(\varphi = s_1) \vee \ldots \vee (\varphi = s_n) \Rightarrow store\,(v, \varphi, f) \quad (4)$$

The raise function can be modeled as:

$$(\varphi = s_1) \vee \ldots \vee (\varphi = s_n) \wedge (select\,(v, \varphi) \neq \eta) \Rightarrow select\,(v, \varphi) \quad (5)$$

3.3 Input / Output Stream Libraries

The libraries group related to flow control of input and output data inside to certain program is represented for the libraries set called Input / Output Stream. Aiming to build the operational model of this set, we elaborated a hierarchical structure really similar to real that can be observed in Figure 2.

3.4 Standard Template Libraries

This set is subdivided in four categories: Algorithms, Numeric, Containers and Iterators. The algorithm library is the only one that composes the Algorithms categories. The operational model of this library is formed by 66 functions that present a lot of algorithms related to manipulation of STL containers, arrays and string objects. The operation of this model can be illustrated by the formalization of the functions $find$ and for_each. The find function looking for a certain element within a range and for_each function applies each element of a range in a certain function. For this, we assume that η be

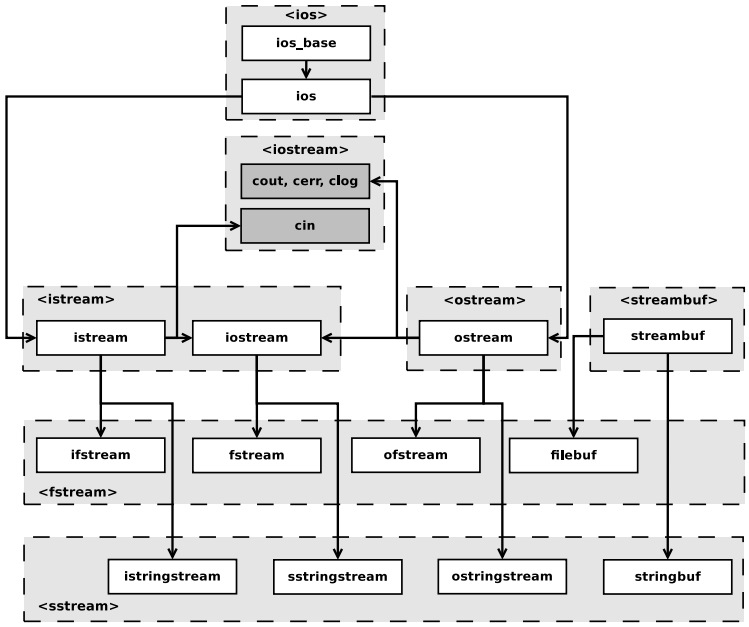

Fig. 2. Diagram about hierarchical structure of the Input / Output Stream Libraries.

a NULL element and I represent a range, such that $\{first, last, n, i\} \subset I$, so that $first$ represent the pointer to beginning of range, $last$ represent the pointer to ending of range, n be the number of elements contained between $first$ and $last$, and i be a certain position within the range. Furthermore, by virtue of formalization of the for_each function we assume f be a function and by virtue of formalization of the $find$ function we assume that α be a non-deterministic value and β be a boolean value that assume $true$ if the element sought is contained in the range and $false$ otherwise. We can thus model the for_each of the algorithm library as:

$$((first \wedge last \wedge f) \neq \eta) \Rightarrow \bigwedge_{i=0}^{n} ((store\,(first, i+1, first_i + 1)) \wedge (f\,(first_i))) \tag{6}$$

The $find$ function can be modeled as:

$$((first \wedge last \wedge \alpha) \neq \eta) \Rightarrow \bigwedge_{i=0}^{n} ((store\,(first, i+1, first_i + 1)) \wedge (ite\,(first_i = \alpha, \beta = 1, \beta = 0))) \tag{7}$$

The operational model created to represent the Numeric category is formed by the numeric library and contains 4 algorithms that work objectively in the manipulation of numerical sequences. One of the algorithms implemented in this model is called $accumulate$, whose main functionality is to accumulate the values belonging a numer-

ical sequence. In order to exemplify the implementation of this model, we assume that α be an accumulator.We can thus model the $accumulate$ of the algorithm library as

$$((first \wedge last \wedge \alpha) \neq \eta) \Rightarrow \bigwedge_{i=0}^{n} ((store\,(first, i+1, first_i + 1)) \wedge (\alpha = \alpha + first_i)) \tag{8}$$

Structure. In this section, we will focus on the STL sequential containers operational model, its preconditions and simulation features (e.g., how we store the elements values of the containers and intern class methods), and how these are used to verify real-world C++ programs. The structure of the STL containers is based on the C++ structure itself, which includes its classes, operators, methods, functions, and intern variables. It is split into: iterations, capacity, element access, modifiers and unique members. As the containers structure differs a little between each other, some of their methods will vary too, changing its intern model as well. For example, a list container does not have a reference operator (i.e., $operator\,[]$), and the elements are reached only by iterators (list has a dynamic structure, unlike deque or vector). Lists also have unique methods, optimized by its dynamic structure such as merging, elements sorting, and reverse order. Vector has also unique methods, related to size and capacity manipulation, and it also does not have $push\_front\,()$ and $pop\_front\,()$ methods [16].

Model Semantics. Let us consider that a container model is composed by five types of variables, which are: I, C, N, P and T. Let I be an iterator that points to a position in the container; C be a container itself; N be a natural integer number used in the container such as size, capacity and elements index; P be the memory address where T is located; and T be the values stored in the container. For convention, we assume that $(c, v, d, l) \subset C, \{i, j, n\} \subset N$ and $\{it1, it2\} \subset I$.

The containers contain an array of elements T and their positions in the memory are represented by pointers P. Therefore, the syntax for the integer expression is:

$$\begin{aligned} Int \;::=\; & N \mid Z \mid C.size \mid C.capacity \mid \\ & Int\;(+ \mid ? \mid\; * \mid ...)\; Int \mid \\ & It\,(+ \mid -)\; It \end{aligned}$$

Where variables included in the containers like *size* and *capacity* return a integer value as well as the arithmetics operations between integer values. Similarly, the syntax for iterator expressions is:

$$It \;::=\; I \mid It(+ \mid -)It \mid C.begin \mid C.end$$

Where $begin$ and end are methods that return iterators, which point to the beginning and the ending of a container, respectively. We also have iterator operations that return iterators as well. For P (memory address values), the syntax is as follows:

$$\begin{aligned} P \;::=\; & p \mid It.pointer \mid C.array \mid \\ & It.source \mid P(+ \mid -)P \end{aligned}$$

Where $pointer$ is a memory address that stores the element in the container pointed by the iterator, $array$ is another memory address that stores the beginning of the container, $source$ is the address that makes the link between the iterator and its pointed container (which stores the container $array$ value).

There is also the pointer return from pointer operations. The syntax for *T* values is the following:

$$T ::= \ t \mid *It \mid *P \mid C_n$$

Where $*It$ is the value stored in the position pointed by a iterator It. Similarly, $*P$ is the value stored in the P position of the memory, and C_n is an element of a container C in the position n.

Model. To simulate appropriately the containers, our model makes use of three variables: a variable P called $array$ that points to the first element of the array, a natural value $size$ that stores the quantity of elements in the container, and a natural value $capacity$ that stores the total capacity of a container (which is valid only for vectors). Note that, as the elements are added in the container (specifically in vectors) and the size grows, the capacity also grows at a rate of $2 * size$, every time the size reaches the capacity value. Similarly, iterators are modeled using three variables: a variable P called ptr, which contains the memory address to the corresponding element T in the container, a variable N called pos, which contains the index value pointed by the iterator in the container, and a variable P called src, which contains the memory address to the first element T stored in the container.

The vector container model has the following structure: $C = \{array, size, capacity\}$, where $array$ is a memory address where the elements are stored in the container, $size$ is the total number of elements in the container, and $capacity$ is the total capacity of the vector, which is simulated internally in the model. The main methods of a vector (and sequential containers, in general) have only three types of operation: *insertion*, *deletion*, and *search*. Methods such as $push\_back\,()$, $pop\_back\,()$, $front\,()$, $back\,()$, $push\_front\,()$, and $pop\_front\,()$ are only a simplified variation of those main methods, which are optimized for some containers (e.g., popping the last element of a stack). To represent the model, consider a container $C\{cont\}$ with a method $cont.insert \rightarrow It$ that returns an iterator result and makes use of an iterator $It\{ipos\}$ that points to the desired insertion position; a template value $T\{val\}$ with the element to be inserted and an integer $N\{qtd\}$ that informs the amount of elements to be inserted in the desired position.

$$\begin{aligned} cont.insert(ipos, val, qtd) \implies & cont'.size = cont.size \\ & \wedge \ *ipos = val \\ & \dots \\ & \wedge \ *(ipos + qtd - 1) = val \end{aligned}$$

There is also another way to represent the insert method. It is possible to insert a sequence of elements in the desired insertion position, using both iterator or pointer bounds. Let $It\{it_0\}$ be an iterator that marks the first element to be inserted, $It\{it_k\}$ be another iterator that points to the first element after the end of the sequence to be inserted in the required position and let $N\{k\}$ be the length of the array $[it_0\, it_k)$.

Therefore, we have:

$$\begin{aligned} cont.insert(ipos, it_0, it_k) \Longrightarrow\ & cont'.size = cont.size + k \\ & \wedge\ *ipos = *it_0 \\ & \ldots \\ & \wedge\ *(ipos + k - 1) = *(it_k - 1) \end{aligned}$$

The same model above is valid for pointers $P\{pt_0\}$ and $P\{pt_k\}$. This kind of insertion (with pointers) does not return a iterator.

The erase method works similarly to the insert method. It also uses iterator positions, integer values and pointers, but it does not use values, as the exclusion is made by a given position, regardless the value. It also returns a iterator position, pointing to the position next to the previously erased part of the container. The next model shows an erase method that deletes a single element:

$$\begin{aligned} cont.erase(ipos) \Longrightarrow\ & cont'.size = cont.size - 1 \\ & \wedge\, ipos' = ipos + 1 \end{aligned}$$

It is also possible to delete a number of elements from the container, marking the bounds with iterators. It works similarly to the equivalent insert method:

$$\begin{aligned} cont.erase(ipos, it_0, it_k) \Longrightarrow\ & cont'.size = cont.size - k \\ & \wedge\, ipos' = it_k \end{aligned}$$

Searches are made in a container by using reference operators and a pointing type (pointer or iterator), and return the reference value (the element stored itself). It can be considered as values $N\{*It\}$, $N\{*P\}$ or $N\{C_n\}$. The structure of iterators is treated differently from other types. The model is the following: $It = \{ptr, n, addr\}$, where ptr is a memory address that points to the real position of the required element in the container (pointed by the iterator), n is the iterator indexed internally in the container, and $addr$ is a memory address equivalent to $cont.array$, being $cont$ the container pointed by the iterator.

3.5 General Libraries

There is a libraries' set that performing more specific functions in the context of the C++ Standard Libraries. In order to build the operational model about this libraries' set we defined a category that represent it. This category is called General Libraries and is subdivide in four particular categories: Language Support, Diagnostics and General Utilities. The Language Support category is compose by four libraries that work respectively with exception handling (exception library), types information (typeinfo library), dynamic allocation of memory (new library) and numeric limits (limits library). The model of each library contains an structure formed by classes, specific types, methods and functions, however the limits library is one exception, seeing that it implements the definitions related to numeric elements, therefore, your structure is formed by one class with the numeric elements definitions and eight functions for manipulate this elements.

The verification of all properties related to libraries is implemented utilizing assertives within each model.A lot of functionalities related to exception handling, dynamic allocation of memory and types information have been implement in the ESBMC's kernel, however is important to have one representation about the libraries relationship to this functionalities so that ESBMC is capable of identify all methods and definitions related to same.

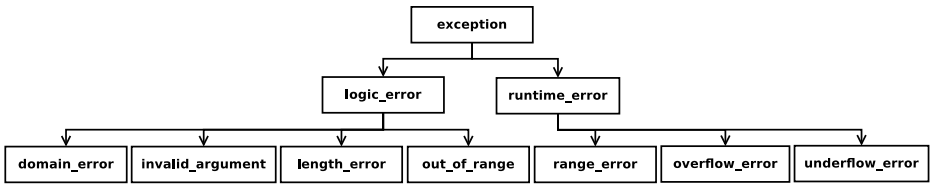

Fig. 3. Diagram about hierarchical structure pf the stdexcept library.

The stdexcept library is the only one the form the Diagnostic category, seeing that it implements kind of standard exceptions divide in two subcategories: logic errors and runtime errors. The relationship between classes that represent the Diagnostic category builded in the operational model is really similiar to real and can be observed in the Figure 3. To build it model, we defined the exception set E assuming that

$$\left\{ \begin{array}{l} logic, domain, invalid\_argument, length, out\_of\_range, \\ runtime, range, overflow, underflow \end{array} \right\} \subset E \qquad (9)$$

furthermore we assume that e represent an element within the set E such that for each e exist one class φ that represent it, being that all φ contains a constructor that sets a message to be thrown if there is any errors during the program execution. In order to identify this errors in the verification, whenever one of the constructors of these classes is called, one $assert(0)$ is throwing, forcing the ESBMC to identify this error and report it in the counterexample. Therefore we can represent the model of the stdexcept library as:

$$(e \in E) \Rightarrow (\forall e)\,(\exists \varphi)\,((e \equiv \varphi) \Rightarrow assert\,(0)) \qquad (10)$$

The General Libraries category also includes definitions about the manipulation of characters sequences, such definitions are implemented in the library string. In the model constructed to represent this library, we implemented it structure through the class called string that contains the main features of the library, and the iterator class containing the representation of the iterator for strings. In general, we define the string object as an array of char, so that all heuristics that must be observed in handling vectors are therefore approached in this model. From of the principle that main operations implemented by this library are restricted to creation, insertion, deletion and search of elements inside to object string, we take as example the creation and search operations to exemplify and formalize the model created for this library. Therefore, firstly

we assume S as the set of string variables, c as character non-deterministic, the set $\{n, i, pos, length\} \subset N$, and we define that s representing a string, the same has two attributes: the string size described by $length$, and a character sequence that represent it described by str_i, such that i represent the position of some character inside to string. Keep this in mind, can be defined a constructor method that creates a string containing n times the character c and, furthermore the find method that search a character within a certain string from a given position described by pos. Therefore, we can model the respective $constructor$ of the string library as:

$$C := \left[\bigvee_{j=0}^{n} str_j = store\left(str_j, i_j, c_0\right) \right] \tag{11}$$

$$P := \begin{bmatrix} (i_0 \geq 0) \ \wedge \ (i_0 < (n_0 + 1)) \\ \dots \\ \wedge \ (i_n \geq 0) \ \wedge \ (i_n < (n_0 + 1)) \\ \wedge \ c_0 \neq 0 \\ \wedge \ n_0 > 0 \end{bmatrix} \tag{12}$$

The $find$ method can be modeled as:

$$C := \begin{bmatrix} (c_0 \neq \phi) \\ \wedge \ (0 \leq pos \leq length) \\ \wedge \ (0 \leq i_n \leq length) \end{bmatrix} \tag{13}$$

$$P := \left[\bigwedge_{j=0}^{length} str_{j+1} = ite\left(select\left(str_j, i_j\right) = c_0, i_j, -1\right) \right] \tag{14}$$

3.6 Inheritance and Polymorphism

Modeling C++ features like inheritance makes static analysis difficult to implement. We highlight the difficult in relation to other object-oriented languages, such as Java, the hierarchy of classes. C++ allows multiple inheritance, when a class may inherit from one or more classes, disallowing the direct use of tools already created for other languages like Java. When a class inherits from a base class features not virtual, this is called replicated inheritance. If a class inherits from a base class virtual, has the name of shared inheritance. A formal description to represent the relationship between the classes can be described by the Class Hierarchy Graph (CHG). This graph G is formed of a tuple $<$ **C**, $\prec s$, $\prec r >$,**C** being the set of classes. $\prec_s \subseteq$ **C** $\times$ **C**, referring to shared inheritance edges and $\prec_r \subseteq$ **C** $\times$ **C**, are replicated inheritance edge. E $\prec_{sr} = \prec_s \bigcup \prec_r$.

We made an intermediate representation, see Figure 5.

Fig. 4. Multiple inheritance: (a) C++ program, (b) class-inheritance graph.

This model transforms all the classes in structures and joins all methods and attributes of its parent classes. This approach has the advantage in verification due the direct access, allowing an easier validation. On the other hand, we replicate information to any new class, wasting memory resources. The indirect inheritance:

Fig. 5. Multiple inheritance intermediate representation.

Fig. 6. Indirect inheritance: (a) C++ program, (b) class-inheritance graph.

3.7 Exception Handling

One of the features that C++ provides over C is exception handling. The exceptions are unexpected situations on the program, situations that the program was not designed to handle. On C++, the exception handling is divided in two blocks: a try block and a catch block.

The try block contains the code that is expected to run without errors and on the catch block, the code to handle errors from the try block. The connection between the two blocks is made by a throw expression. The throw statement is an expression of type void that invokes the handler from inside the try block and accepts one parameter, which is passed as parameter to the handler.

The exception handling brings a lot of advantages to the C++ development, because it allows the separation between code and error handling code. Futhermore, the exceptions propagates up in the stack, meaning that if a C++ program calls several functions, only one of than needs to handle the exceptions thrown.

Throwing an Catching an Exception. There are also other ways to an exception be thrown by an program, other than the throw statement, such as the new operator can throw bad_alloc exception, dynamic_cast can throw bad_cast exception and typeid can throw bad_typeid exception. Also those exception are builtin on C++ and are supposed to be handled by the programmer during the development.

When throwing an exception the control and all the information are transferred to the handler. The rules to connect the exception with its handler are defined on the C++ standard draft [15] as follow:

- The handler that will catch the exception will be the first catch with a matching type.
- A handler will catch a exception thrown if the type of the throw and the type of the handler are the same (ignoring const-volatile qualifiers.
- Throwing ”arrays of type T” and ”functions returning type T” with be handled by handles with ”pointer to type T” and ”pointer to function returning type T” types.
- The handler will catch an exception of type T if the handler type is an unambiguous public base class of T.
- The handler will catch an exception of type pointer T if Ts type can be converted to the type of the handler, either by qualification conversion or standard pointer conversion.
- If the exception throw is a pointer then a handler with type void* or nullptr_t can also catch it.
- A handler of type ellipsis (...) will catch any thrown exception, and shall be the last handler on the catch block.

Fig. 7. Indirect inheritance intermediate representation.

```
int main() {
  // try block
  try {
    throw 20;
  }
  // catch block
  catch (int) { /* error handling */ }
  return 0;
}
```

Fig. 8. Try-catch example: Throwing an int type.

Rethrows and Exception Specifications. C++ also provides features as rethrows and exception specifications. The rethrows allows us to rethrow the last thrown exception, as shown on Figure 9. Rethrows are useful for propagating the exception up in stacks with lots of try-catch blocks.

```
int main() {
  try {
    throw 20;
  }
  catch(...)  {
    throw; // Rethrows type int
  }
  return 0;
}
```

Fig. 9. Try-catch example: Rethrowing an int type.

Exception specifications define which exceptions a function or method can throw. Figure 10 shows an example of exception specifications usage. They are useful when we want to forbid that certain types of exceptions can be thrown, for example, we can define that an function or method cannot throw any exception.

4 Experimental Results

Experimental Results here (Mauro)...

```
1  // This function can only throw int types
2  void x() throw (int) {
3    throw 20;
4  }
5  int main() {
6    try {
7      x()
8    }
9    catch(...)  {
10     throw; // Rethrows type int
11   }
12   return 0;
13 }
```

Fig. 10. Try-catch example: Allowing that only int type can be thrown.

5 Related Work

Prabhu et al. presents an interprocedural exception analysis and transformation framework for C++ that records the control-flow created by the exceptions and create a exception-free program. The exception-free program creation starts by generating a modular interprocedural exception control-flow graph (IECFG). The IECFG is refined using a algorithm based on a compact representation for a set of types called the Signed-TypeSet domain and the result is used to generate the exception-free program. Finally, the exception-free program is verified using F-SOFT [17]. The verification focused on two properties: ”no throw”, the percentage of the code that does not raise an exception and ”no leak”, the number memory leaks on try-catch blocks. [2]

6 Conclusions

Conclusions here (Lucas)...

Acknowledgments. Acknowledgments here (Lucas)...

References

1. R. L. Sites, “Some thoughts on proving clean termination of programs.” Stanford, CA, USA, Tech. Rep., 1974.
2. P. Prabhu, N. Maeda, G. Balakrishnan, F. Ivancic, and A. Gupta, Interprocedural Exception Analysis for C++. In *ECOOP*, pp. 583–608. 2011.
3. N. Blanc, A. Groce, and D. Kroening, Verifying C++ with STL containers via predicate abstraction. In *ASE*, pp. 521–524. 2007.
4. A. Biere. Bounded model checking. In *Handbook of Satisfiability*, pp. 457–481. 2009.
5. A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: a software model checking approach. *FMCAD*, 2010, pp. 121–128.
6. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *TACAS*, *LNCS* 2988, pp. 168–176, 2004.

7. L. Cordeiro. SMT-Based Bounded Model Checking of Multi-Threaded Software in Embedded Systems. PhD Thesis, U Southampton, 2011.
8. L. Cordeiro and B. Fischer. Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. *ICSE*, pp. 331–340, 2011.
9. J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Context-Bounded Model Checking of LTL Properties for ANSI-C Software. *SEFM*, *LNCS* 7041, pp. 302–317, 2011.
10. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, vol. 11 (1), pp. 69–83, 2009.
11. M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. *ICCAD*, pp. 794–801, 2006.
12. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, v. 38, n. 4, pp. 957–974, 2012.
13. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. *VSTTE*, pp. 146–161, 2012.
14. OpenC++, http://opencxx.sourceforge.net/, 2012.
15. C.standards commitee. Working draft, standard for programming language C++ (2012), http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2012/n3376.pdf (acessed 8 october, 2012)
16. Reference of the C++ Language Library, http://www.cplusplus.com/reference/, 2012.
17. F. Ivancic, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, Z. Yang. Model Checking C programs using F-Soft. *IEEE International Conference on Computer Design*. pp. 297–308, October, 2005.