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 C programs due to features that the languages offers, such as templates, containers and exception handling. We present ESBMC, a bounded model checker for C and C++ programs, which encodes the verification conditions using different background theories and passes them directly to an SMT Solver. Our experimental results show that our approach can handle more constructs of the C++ programs and substantially reduce the verification time.

1 Introduction

Introduction here (Lucas)...

2 Background

Background here (Lucas)...

3 SMT-Based BMC for C++ Software

4 Standard Template Libraries

4.1 Introduction

Several applications have C++ as its basic programming language, including verification tools, information retrieval machines, databases, simulators, embedded systems and telecommunication systems. Compared to C, this programming language provides a wider set of features, libraries and functionalities that would require a way larger codification in the basic version of this language. This features include OOP (Object-oriented programming), specialized input-output libraries (stream libraries, in general), the template concept ,including STL containers, which have some of the popular data structures in Computing Science, plus making possible the use of any data type, thanks to template usage.

Systems that make use of this language (and its functionalities) require strong and steady software verification, for its errors become harder to find, as its own structure grows.

For this kind of operation, verification software is becoming more important and critical, especially its crucial characteristics (speed, accuracy, efficiency, friendliness). Many verifying systems are available, especially in high-level test. In a particular approach, STL verification become very useful as the system needs grows, for its practicality and encapsulated features accelerates the programming process, shortening the auxiliary implementation (like data structures creation, sort algorithms, search functions), helping the programmer focus on what matters.

Of course, if C++ seems to be a more complex version of C, its verification will be more complex as well. Our solution involves the usual C code verification, using SMT-based bounded model-checking (ESBMC), and an operational model of the C++ libraries. In this session, we will focus on the STL sequential containers operational model, its preconditions and simulation features (like how we store the elements values of the containers and intern class methods), and how this is used to verify a C++ program code.

4.2 Structure

The structure of STL containers is based on the C++ structure itself, including its classes, operators, methods, functions and intern variables. It is divided in: iterations, capacity, element access, modifiers and unique members. The similarities and differences between sequential containers are described at Table 1.

Table 1. STL sequential containers

4.3 Model Semantics

Let us consider that a container model is composed by five types of variables, *I*, *C*, *N*, *P* and *T*. *I* represents a iterator that points to a position in the container, *C* represents the container itself, N represents natural integer numbers used in the container, like size, capacity and elements index, *P* represents the memory address where *T* is located, and *T* represents de 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*. Assuming this, the syntax for integer expression is:

Int = $N \,|Z\, |C.size\, |C.capacity\, |$*Int*$(+\,|?\,|*\,|...)$*Int*
$|$*It*$(+\,|-)$*It*

4.4 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 elements array, a natural value size, that stores the quantity of elements contained in the container, and a natural value capacity, that stores the total capacity of a container (valid onliy 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 representing variables: a variable P called pointer, which contains the memory address to the correspondent element *T* in the container, a variable N called position, which contains the index value pointed by the iterator, in the container, and a variable P called source, which contains the memory address correspondent to the first element *T* stored in the container.

The vector container model has a structure as it follows:

C1 = {P1, C1.size, C1.capacity}

Where P1 is a memory address where it is stored the elements of the container, C1.size is the total number of elements in the container, and C1.capacity is the total capacity of the vector, simulated internally in the model.

The main methods of a vector (and sequential containers, in general) have only three types of operation: insertion, exclusion and search. Methods like push-back(), pop-back(), front(), back(), push-front() and pop-front() are only a simplified variation of those main methods, optimized for some containers (like pop-back in a stack).

An insert method is represented by the following structure:

Ccont
Itcont.insert—cont.insert
Itposition Itfirst Itlast
Ppoint1 Ppoint2
Tvalue
Nquantity
cont.insert (position, value)
cont'.size = cont.size + 1
position = value
—cont.insert(position, value, quantity)
cont'.size = cont.size + quantity
(position + N(0 -¿ quantity)) = value
—insert(position, first, last)
cont'.size = cont.size +(last - first)
*position = *first*

—insert(position, point1, point2)
cont'size = cont.size + (point2 - point1)
position = *point1

Similarly, an erase method is structured like the following:

Ccont
Itcont.erase—cont. erase
Itposition Itfirst Itlast
Ppoint1 Ppoint2
cont. erase (position)
cont'.size = cont.size - 1
position' = position + 1
—cont. erase (position, first, last)
cont'.size = cont.size -(last - first)
position' = last

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).

Ccont
Iti
Nn
TC[n]
T*i

The structure of iterators is treated differently from the other types. The model is the following:

It : Ppointer; Npos; Pcont$_{p}os$

Where pointer is a memory address that points to the real position of the required element in the container (pointed by the iterator), pos is the iterator index internally in the container, and cont$_{p}osisamemoryaddressequivalenttocont.buf, beingcontthecontainerpointedbytheiterator$

5 Experimental Results

Experimental Results here (Mauro)...

6 Related Work

Related Work here (Mauro and Mikhail)

7 Conclusions

Conclusions here (Lucas)...

Acknowledgments. *Acknowledgments here (Lucas)...*

References

1. *A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: a software model checking approach.* FMCAD*, 2010, pp. 121–128.*
2. *E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs.* TACAS, LNCS *2988, pp. 168–176, 2004.*
3. *L. Cordeiro. SMT-Based Bounded Model Checking of Multi-Threaded Software in Embedded Systems. PhD Thesis, U Southampton, 2011.*
4. *L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software.* ASE*, pp. 137–148, 2009.*
5. *L. Cordeiro and B. Fischer. Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking.* ICSE*, pp. 331–340, 2011.*
6. *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.*