Science of Computer Programming 128 (2016) 68–85

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Model checking C++ programs with exceptions ☆

P. Ročkai *,[1], J. Barnat, L. Brim

Faculty of Informatics, Masaryk University, Botanická 68a, Brno, Czech Republic

ARTICLE INFO

Article history:
Received 2 May 2015
Received in revised form 11 May 2016
Accepted 11 May 2016
Available online 31 May 2016

Keywords:
Model checking
C++
Exception handling
LLVM

ABSTRACT

We present an extension of the DIVINE software model checker to support programs with exception handling. The extension consists of two parts, a language-neutral implementation of the LLVM exception-handling instructions, and an adaptation of the C++ runtime for the DIVINE/LLVM exception model. This constitutes an important step towards support of both the full C++ specification and towards verification of real-world C++ programs using a software model checker. Additionally, we show how these extensions can be used to elegantly implement other features with non-local control transfer, most importantly the `longjmp` function in C.

1. Introduction

Enabling the adoption of formal verification methods by the broader software development community is a major goal in computer science research. Applicability of verification methods in realistic scenarios has always been an important benchmark for new methods and tools alike. This trend is even more pronounced in recent work, as exemplified by the activity in the program analysis community and the Software Verification Competition [1]. In terms of adoption, a crucial factor in determining the success or failure of a verification tool (or more generally, a verification method) is its ease of use. It is therefore important to apply formal methods at the appropriate level; extensive planning and design stages are severely reduced or entirely absent in many software projects. With the bulk of software development shifting towards the implementation, the use of traditional model checkers operating on the level of dedicated high-level modelling languages becomes much harder. On the other hand, tools based on existing programming languages, with the ability to work with existing source code, can fit quite naturally into “implementation-heavy” processes.

Besides easy integration into prevalent development workflows, use of source code as the principal input into the verification process has another important advantage: software developers are fluent in the implementation language. This means they can readily and naturally express ideas in that particular language; this is also reflected in the integration of planning and design phases with implementation: software prototyping often happens in the final implementation language, and smoothly progresses into implementation through gradual refinement. A verification tool which can process that particular implementation language can be readily used throughout the process starting from early prototypes (where it can primarily help in ensuring the soundness of the design), ending with the final implementation. Especially towards the later stages, implementation-level properties and become more important, as does fidelity of the verifier with regard to the “real” behaviour of the system.

☆ This work has been partially supported by the Czech Science Foundation grant No. 15-08772S.
* Corresponding author.
E-mail addresses: xrockai@fi.muni.cz (P. Ročkai), barnat@fi.muni.cz (J. Barnat), brim@fi.muni.cz (L. Brim).
[1] Petr Ročkai has been partially supported by Red Hat, Inc.

http://dx.doi.org/10.1016/j.scico.2016.05.007
0167-6423/

Finally, for a formal system, it is important that the language used for specifying both the system and the requisite properties is precise and unambiguous. In practice, this is always a compromise.[2] While the semantics of programming languages are usually not rigorously specified, they do often attain a very high level of precision in their specifications. In the case of C++, the main inconvenience of the specification is the large volume of text, and consequently, the large number of facts contained therein. Nonetheless, the complex natural-language specs have a more formal counterpart: compilers. While a C++ compiler is a very complex software system, the fact is that real-world compilers achieve a very high level of agreement in their semantics (as compared to each other).

Given those observations, there is a natural tendency to build model checkers [2] that can be applied to programs written in commonly-used programming languages: most importantly C, C++ and Java. Clearly, there are limitations to what a model checker can do: the problem it is tackling is, in general, undecidable. In theory, this is a red flag – we are trying to solve a problem that we know for a fact cannot be solved. Nevertheless, a partial solution can still be very useful: after all, a software engineer often has to argue about properties of programs that are in general undecidable. In this case, all that matters is whether the instance at hand can be solved.

There is, however, another limitation, which is usually more important in practice: conformance to programming language specifications. In order to derive substantial utility from a model checker, it should implement a full programming language specification: the programs that software developers write and which they can run should be also valid inputs to a model checker. This is especially critical if we expect seamless integration of model checking tools into a development workflow. Programming languages, as specified, are already very constraining – engineers in pursuit of more elegant and more maintainable code often approach the boundaries of what is allowed in a particular programming language.[3]

This is especially a concern with C++, which is a relatively high-level language, with a long development history and in widespread use. Some of the features the language offers are quite tricky (especially so in the context of model checking), usually because they exhibit very complex semantics. While some of the problematic aspects can be solved by targeting a suitable intermediate language and repurposing a good existing compiler frontend – such as LLVM [3] (the IR) and CLang (the frontend) – this is not the case with all such features. One particular example is exception handling, which must be retained in the intermediate representation.

Besides their complicated semantics, exceptions bring an entirely new problem to model checking: non-local transfer of control. While not insurmountable, it makes everything more complicated – and a modern software model checker is already complicated enough. However, for the reasons expounded earlier, we firmly believe that it is very important to provide full coverage of language features in a model checker. This paper primarily presents our experience in implementing exception handling in DIVINE, an explicit-state model checker for C and C++ programs based on LLVM.

The remainder of this paper is organised as follows: Section 2 introduces DIVINE and its relation to LLVM and C++, Section 3 gives an overview of related work, while Section 4 discusses the use cases for exception support in a model checker. Section 5 gives an overview of the implementation-level mechanisms which play a central role in exception handling, with focus on LLVM and C++. Sections 6 and 7 discuss the LLVM and C++ support in DIVINE, respectively. Finally, Section 8 is concerned with our implementation and Section 9 gives both a qualitative and a quantitative evaluation of our work. Section 10 concludes the paper.

2. Preliminaries

DIVINE [4] is a general-purpose explicit-state model checker for safety and LTL properties. For LTL model checking, it uses an automata-based approach [5], reducing the decision procedure to a graph problem – namely detection of an accepting cycle in the state space graph of a program under verification. In order to tackle large graphs, it implements efficient parallel algorithms for both reachability (for safety verification) and accepting cycle detection (for LTL model checking). Implementations tailored for both shared-memory and distributed-memory parallel computers are available, along with an assortment of memory-saving techniques. For recent results in the field of parallel and distributed model checking, see e.g. [6].

Among other input languages, DIVINE can handle programs written in the LLVM intermediate representation (LLVM IR). The main use-case for explicit-state model checking, and especially LTL model checking in this area is for unit testing of parallel programs. While explicit-state model checking per se (without the aid of some form of abstraction) cannot handle arbitrary IO behaviour, this is something that software engineers deal with all the time – testing cannot do that either. Of course, an ideal solution would overcome this problem as well – but we contend that this is not a serious obstacle in pragmatic use. However, there are two interesting things that an explicit-state model checker *can* do (and where testing struggles): asynchronous lock-based parallelism (which is an ubiquitous concern in contemporary C++ programming) and liveness (LTL) checking. Moreover, since LLVM is quickly becoming the lingua franca of software analysis tools [7,8], it is

[2] While the compromise is partly dictated by practical considerations, there are important theoretical constraints on the "formality" of any language: for specifying a language, a meta-language is required, and if this meta-language is to be formally specified, infinite regress results.

[3] There are specialised projects where programming language semantics need to be severely constrained, whether it is due to formal treatment – this is sometimes the case with mission-critical software – or due to limitations of the hardware platform, a situation most often encountered in the embedded systems space. In the latter category, however, increases in hardware capabilities of embedded systems are apt to reduce the gap between embedded and mainstream general-purpose programming.