Electronic Communications of the EASST
Volume 70 (2014)

Proceedings of the
14th International Workshop on
Automated Verification of Critical Systems (AVoCS 2014)

Model Checking C++ with Exceptions

P. Ročkai, J. Barnat and L. Brim

15 pages

Guest Editors: Marieke Huisman, Jaco van de Pol
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Model Checking C++ with Exceptions

P. Ročkai[*], J. Barnat and L. Brim

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xrockai,barnat,brim}@fi.muni.cz

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.

Keywords: model checking, C++, exception handling, LLVM

1 Introduction

Widespread and regular use of formal verification methods in the general software development is one of the major goals in computer science. As a matter of fact, recent formal method research trends put a strong emphasis on direct practical applicability of verification results. A current example of this trend is the activity in the program analysis community and the Software Verification Competition [Bey14]. The strong drive to make formal method applications approachable by the general software development and engineering community highlights the fact that the most important factor of using formal methods in practice is their ease of use. Hence, formal methods must be applied at a level that software engineers and developers naturally work at – that is, in an overwhelming majority of cases, at the source code level.

There are multiple reason for this: not only working with program source code is natural for a software developer, source code also constitutes very precise notation, which is a natural match for a formal system. While it is true that semantics of programming languages are usually not rigorously specified, they do often attain a very high level of precision in their specifications. The main inconvenience of specifications of languages like C++ lies in the large volume of text, and consequently, large amount of facts. Nevertheless, the complex natural-language specs have a formal counterpart: compilers. While a C++ compiler is a very complex software system, the fact is that real-world compilers achieve an enviable level of agreement in their semantics, despite numerous optimisation passes they all implement.

Consequently, there is a natural tendency to build model checkers that can be applied to programs written in commonly-used languages: most importantly C, C++ and Java. Clearly, there

[*] Petr Ročkai has been partially supported by Red Hat, Inc.

are limitations to what a model checker can do: the problem it is tackling is, in general, firmly undecidable. In theory, this is a huge 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 immensely 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 that they can run should be also valid inputs to a model checker. This is especially critical if we expect a seamless integration of model checking tools into a development workflow. The brunt of the problem at hand is that programming languages as specified are already very constraining – engineers in pursuit of more elegant and more maintainable code already skirt the boundaries of what is allowed in a particular programming language. Introducing substantial constraints to enable model checking is, in many cases, a non-starter.[1].

This is especially a concern with C++, which is a relatively high-level language, with a long development history and widespread use. Some of the features the language offers are rather unpalatable (especially so in the model checking community), usually because they exhibit very complex semantics. While some of the problematic aspects can be conjured away by targeting a suitable intermediate language and hijacking a good existing compiler frontend – such as LLVM [LA04] (the IR) and its companion CLang (the frontend) – this is not the case with all such the features. A particularly hairy example is exception handling, which necessarily finds its way into the intermediate representation.

Besides their complicated semantics, which are already a formidable problem, they bring an entirely new phenomenon to model checking: non-local transfer of control. While not unapproachable, it complicates everything – and a modern software model checker is already complicated enough. It is easy to see how tempting it is to constrain the input language of the model checker to disallow exceptions. 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.

2 Preliminaries

DIVINE [BBH+13] is a general-purpose explicit-state model checker for safety and LTL properties. For LTL model checking, it uses an automata-based approach [VW86], 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

[1] Clearly, 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. Nevertheless, in the latter category, increases in hardware capabilities of embedded systems is apt to reduce this gap between embedded and mainstream general-purpose programming.

model checking). Implementations tailored for both shared-memory and distributed-memory parallel computers are available, along with an assortment of memory-saving techniques. For more recent results in the field of parallel and distributed model checking, see e.g. [ELPP12].

Among other input languages, DIVINE can handle programs written in LLVM intermediate representaion (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 [MFS12, CDE08], it is not unconceivable that an explicit-state model checker would be integrated into a larger abstraction/CEGAR-based decision procedure, in order to tackle the open-world angle in program verification.

Now if we have a model checker that can handle LLVM IR as its input, and a compiler frontend that can translate C++ into LLVM IR (and there are at least two such compilers, CLang and GCC) – we can compile C++ programs to IR (bitcode) and run a model checker on it. This appears to be very easy on the surface of it, but there are hidden complexities. First, in order to actually verify a program, it needs to be fully defined: in basically all cases, programs make use of the C and C++ standard libraries. Especially the C++ standard library constitutes a very substantial amount of code, and we cannot reasonably argue about program correctness if we don't include this code. The libraries in turn make use of system-level APIs (in the cases we are interested in, this is mostly POSIX), which are fortunately fairly constrained and small, compared to the standard libraries themselves.

As we have argued above in Section 1, constraining the language that can be used with the model checker is an option of last resort, as it has substantial impact on its usefulness. A good way to approach this problem is to compile the libraries themselves into bitcode and bundle them with the program's own bitcode, to form a nearly fully-defined LLVM program. The missing system-level APIs can be either provided as stubs (a practice commonly used in testing), or in some cases – where they constitute important part of program functionality – implemented in terms of a small number of model-checker provided primitives. To this end, DIVINE provides an implementation of both the C and C++ standard libraries, including the small number of changes required for compatibility with the different system-level API implemented by DIVINE. The overall structure of the libraries, with emphasis on exception handling (see also Section 3 for more details on this) is illustrated in Figure 1.

3 Exception Handling

Exception handling is an area where code generation needs to co-operate in order to implement correct language semantics. Since code generators are part of LLVM, but LLVM itself is programming-language-agnostic, the LLVM code generators need to provide a sufficiently generic

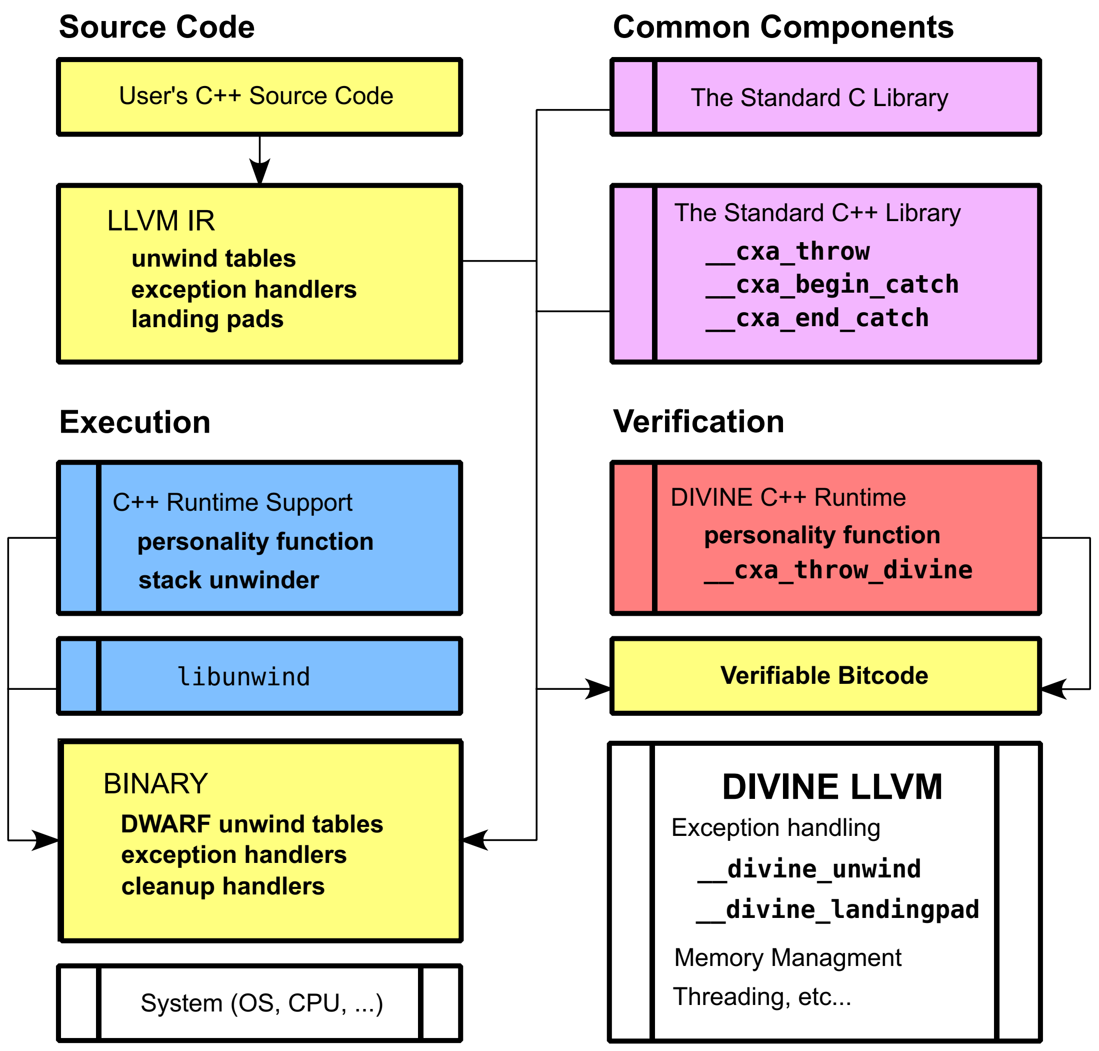

Figure 1: The various components involved in exception handling, and their interaction with execution and verification. The source code is first compiled using a suitable C++ frontend (clang or gcc) into LLVM IR. When building a binary for execution, the IR code is fed to a code generator and combined with common components (the standard C and C++ libraries), and with execution-specific components: `libunwind` and execution-specific parts of the C++ runtime support library (the personality routine and the `libunwind`-based stack unwinder). For verification purposes, the LLVM IR is instead combined with those same common components that have been converted into intermediate representation, and with verification-specific runtime functions from the DIVINE C++ runtime. The resulting bitcode file is then fed to DIVINE, using its LLVM subsystem to generate the state space and execute a suitable verification algorithm on that state space.

interface to allow implementation of efficient exception handling.

In all modern C++ compilers, zero-cost exceptions are the norm: the exception handling machinery imposes no overhead at all unless an exception is actually thrown. This means that the code generator is not allowed to insert special instructions for calls or for saving context when entering try blocks. In order to allow this sort of behaviour, all exception handling logic needs to happen at an exception throw time, and for this to be possible, a stack unwinder is required. The unwinder is platform-specific, and needs to understand the particular ABI and most importantly the layout of the program stack and individual stack frames. LLVM itself does not provide an unwinder library: it is usually provided by the operating system.

Unfortunately, the interface of the unwinder library is not entirely specified, and as such, it is also somewhat platform-specific. There are two major surfaces of the unwinder, each exposed to different part of the compiler/standard library duo. On one hand, the unwinder needs *unwind tables* in order to correctly unwind the stack. These unwind tables are generated by LLVM, since they reflect the high-level structure of individual stack frames, which is itself generated by LLVM. These tables end up being a part of the *program text*, i.e. they are stored in the executable image, and are as such a static part of the program. On the other hand, there is the “dynamic”, or runtime, interface of the unwinder library, which is exposed to the language runtime instead: when an exception is raised, the language runtime uses the unwind library and the unwind tables generated by LLVM to guide the exception handling process.

While C++ is the primary target of the exception-handling mechanisms in LLVM, care has been taken to make it sufficiently general to accommodate other language runtimes, as long as their exception handling works along the same general principles. The main requirement for an exception system to be compatible with LLVM is that it can use the same unwinder interface, or at very least that it can process the unwind tables produced by LLVM. On many platforms (all modern UNIX systems based on the ELF executable format), these unwind tables are in a standardised format, mandated by the DWARF specification [DWA10]. Other platforms use different unwind tables, though.

Besides information about the structure of a stack frame, unwind tables contain information about how exception handling should process this particular stack frame. In programming languages with lexical scoping, lexically scoped variables cease to exist when their scope terminates: normally, this happens when a function returns. However, exceptions create a new way in which a lexical scope can cease to exist, namely that an exception is propagated through this scope upwards. As long as lexically scoped (local) variables are sufficiently simple (plain old data in C++ terminology), this is not a major problem: the stack is unwound, so the storage associated with those variables is automatically reclaimed. However, C++ and a number of other languages allows scoped variables of complex types, with associated destructors: code that the runtime guarantees is executed just before the variable is deallocated. Particularly in C++, this is widely used to implement reliable, automatic resource acquisition and release[2]. Even though similar schemes have been proposed for C [Tur], they are usually implemented using `setjmp` and `longjmp` primitives, do not use any compiler support and therefore do not map to the LLVM

[2] In the C++ community, this design pattern is known as RAII: Resource Acquisition Is Instantiation. Among other things, it is used to safely hold mutual exclusion locks, dynamically allocated memory and other non-composable resources inside functions that could experience non-local loss of control due to exceptions.

exception handling mechanism.

Nevertheless, LLVM as such has no concept of destructors, nor does the unwinder library. The language compiler needs to generate *cleanup handlers*, i.e. blocks of code that take care of calling any appropriate destructors, or performing other language-specific cleanup when a stack frame is torn down because the stack is being unwound. Moreover, the same mechanism is used for *exception handlers*: the main difference is that an exception handler *stops* the propagation of an exception, and its role is to deal with the exceptional situation: exception handlers correspond to the *catch* blocks attached to a *try* block.

In order to improve efficiency (at the expense of simplicity) of the unwinder, it has a concept of *exception type*: different types of exceptions can happen, and a particular catch block can handle only a subset of those exception types. Each call-site in each call frame possibly contains a cleanup handler, and a list of exception handlers. Deciding whether a particular exception handler can handle a particular exception type is deferred to a *personality function*: a language-specific callback provided to the unwinder. This personality function helps the unwinder decide, among other things, which handler to invoke for a particular exception type.

3.1 Mapping Exceptions to LLVM

Now that we have established the basics of how exceptions are implemented in general, we will look at how those concepts map to LLVM. The machinery provided by LLVM to handle exceptions consists of 3 instructions: `invoke`, `landingpad` and `resume`. The `invoke` instruction is like a `call` instruction, but it provides extra provisions for exception propagation: unlike `call`, it is a terminator instruction, i.e. it is always last in a basic block. It is also a branching instruction: it takes two basic block addresses as parameters corresponding to two branches – the first is taken upon a normal return from the function, the other is taken if an exception has been raised in the callee.

The `invoke` instruction co-operates tightly with the `landingpad` instruction: the basic block that the exception branch of `invoke` points to must begin (after any possible φ instructions) with a `landingpad` instruction, and the entire basic block is called a *landing block*[3]. The `landingpad` instruction then encodes the list of exception handlers and whether there is a cleanup handler present, and which personality function to invoke for the corresponding callsite (`invoke` instruction). The syntax of the `landingpad` instruction is following:

```
<r> = landingpad <rt> personality <t> <pers_fn> <clause>+
<r> = landingpad <rt> personality <t> <pers_fn> cleanup <clause>*

<clause> := catch <type> <value>
<clause> := filter <array constant type> <array constant>
```

If the landing block is a cleanup one, the stack unwinder always transfers control to the landing block during the unwinding process, regardless of any exception handlers. If the landing block is

[3] In upstream LLVM documentation, what we call a “landing block” here is referred to as a “landing pad”. The reason for this departure is that the original terminology makes it easy to confuse “`landingpad`” as an instruction and “landing pad” as a basic block.

not a cleanup landing block, it is only executed if some *catch clause* in the *landingpad* instruction matches the exception type (as decided by the provided personality function).[4]

Since each *invoke* instruction only has a single landing block associated, this landing block is responsible for handling any and all *catch* clauses of the higher-level programming language covering the particular callsite. The return value of the `landingpad` instruction is crucial in deciding what action to take when the landing block is entered, and corresponds to the return value of the personality function. In other words, when the unwinder executes the personality function (which is part of the language runtime), it stores its return value, and provides this return value in the result of the `landingpad` instruction. Since the personality function has access to the part of the unwind tables generated from the `landingpad` instruction, it can communicate information encoded in the unwind table to the landing block itself. In the libc++ runtime, the personality function returns a tuple consisting of a pointer to the exception object itself, and a "handler switch value", an integer which corresponds to the index of a relevant "catch" clause of the `landingpad` instruction, or a special value (-1) when no catch clauses match but a cleanup needs to be performed.

The code generated for the landing block then checks the handler switch value computed by the personality function, and transfers control to a cleanup or handler block accordingly. Finally, if the selected handler is a cleanup handler, the exception propagation (stack unwinding) needs to be resumed after the cleanup is done. This is achieved by the `resume` instruction, which expects as a parameter the same value that was returned by the corresponding `landingpad` instruction which interrupted the exception propagation.

Interestingly, there are no LLVM instructions for *raising* (throwing) exceptions. This is left entirely in the management of the language runtime, which needs to closely co-operate with the stack unwinding library anyway (the interface of the personality function is mandated by the stack unwinder).

4 C and C++ Runtime Support in DIVINE

As we have argued in Section 2, in order to verify real-world code, we need to provide an implementation of standard libraries: DIVINE provides `libc`, in form of bitcode that can be linked to (incomplete) bitcode produced by the compiler from the C program itself. While the implementation of `libc` is mostly complete, in some respects, it behaves differently from traditional OS-provided versions. Since the program that is being verified is not allowed to actually interact with the world, such function calls are implemented either as "stubs" possibly using non-deterministic choice, or they interact with DIVINE using a private DIVINE-specific interface.

The case of C++ is slightly more complicated. While many language features require no special runtime support (i.e. the same as C), there are some that do, most notably Run-Time Type Identification (RTTI) and exception handling. Besides those areas where library support code is required for *language* features, like in C, most C++ programs make use of the standard

[4] Additionally, the filter clauses restrict the types of exceptions that can be propagated through the `invoke` instruction corresponding to this landing block, akin to how *exception specifiers* work in C++. If an exception is thrown and it reaches a filter clause of the appropriate type, a language-specific action is invoked. In C++, this action is user-specified, and defaults to terminating the program.

C++ library.

Consequently, there are two libraries that are required by virtually all C++ programs: the runtime support library, and the standard library. Multiple implementations of both exist[5] – DIVINE ships with `libc++abi` for the runtime portion and `libc++` for the stdlib portion.

As far as RTTI goes, there are no special considerations with regards to model checking. The upstream `libc++abi` code can be used verbatim with DIVINE. Exceptions are more complicated, and are, coincidentally, a feature that is most often neglected in analysis tools and model checkers that work with C++ programs. Exception handling in C++ consists of three major parts: unwind tables, landing pads and exception handlers which are all generated by the compiler based on the input code, using special (although language-neutral) LLVM instructions: `invoke` and `landingpad` being the two most notable. Additionally, the C++ runtime library uses a CPU- and platform-specific *stack unwinder* and contains a language-specific *personality routine*. The personality routine makes use of the unwind tables generated by the compiler to guide the stack unwinder during an exception (see Section 3 for details).

An LLVM interpreter hence needs to provide a stack unwinder and an API to access the unwind tables, for use by the personality routine. In DIVINE, the unwinder interface is extremely simple, consisting of a single trap, `__divine_unwind`. The language runtime can use `__divine_unwind` to remove a number of topmost stack frames from the stack of the current thread, returning control to the topmost remaining frame. If the active instruction in the target frame is an `invoke` instruction, control is transferred to its alternate destination basic block (a landing block), and the value passed to `__divine_unwind` is passed on to the personality routine of the landing block.

5 Implementation

We have outlined the mechanisms used by LLVM to implement language-agnostic exception handling in Sections 3 and 3.1. There are multiple points where DIVINE has to hook into those mechanisms in order to support exception handling in a particular programming language. While a substantial part of that support is language-agnostic, crucial pieces of infrastructure are part of the language's standard library: in case of C++, this is `libc++abi` as explained in Section 4.

In a native code generator in LLVM, the information from `landingpad` instructions generated in the frontend is used to construct unwind tables. The format of those tables is platform- and architecture- specific. To read those tables, `libc++abi` uses the `libunwind` interface (originally specified as part of the IA64 C++ ABI). This interface is semi-standard, but no actual standardising document exists. Since the `libunwind` implementation is tied to the binary format of the executable, via the in-memory image of the unwind tables, it cannot be directly used in DIVINE. Likewise, it is tied to a specific architecture/platform via its knowledge of stack and register layout – another disqualifying feature. Therefore, `libunwind` needed to be replaced with a new implementation for DIVINE.

[5] The GNU compilers ship with `libstdc++`, which contains, as a subproject a runtime support library libsupc++. Clang ships with `libc++`. Depending on platform, a choice of either `libc++abi` or `libcxxrt` is available for use with `libc++`. An independent implementation is available from Apache Software Foundation under the name `libcxx`. Multiple compilers ship yet different implementations.

```
struct R1 { R1() { /* ... */ } ~R1() { /* ... */ } };
struct RC { R1 r1; int *resource;
  RC() : r1() {
    resource = new int[32];
  }
  ~RC() { delete[] resource; }
};

int main() {
  try {
    RC res;
    // work with the resource...
  } catch (...) {
    // handle exceptions
  }
}
```

Figure 2: Example source code, for illustrating exception handling mechanisms. See Figures 3 and 4.

5.1 The `libunwind` interface

There were two basic options: either replicate the portion of the `libunwind` interface used by `libc++abi`, making it possible to use unmodified source for `libc++abi` – which sits on a higher level than `libunwind`. Conceptually, this is a tempting solution – the more of the library code is left intact, the more faithful the verification. There is a major downside though: the interface between `libunwind` and `libc++abi` is complex and intricate. This is especially true of the interface between the unwinder and the personality function: the unwinder uses the personality function as a callback, invoking it once for each active frame on the stack at the moment an exception is raised. The personality function uses a pair of platform-specific registers to pass the handler switch value and the exception pointer to the exception handler: it cannot invoke the handler itself, as the stack has not been unwound yet and the handler would end up running in the wrong context. For this reason, `libunwind` provides an interface to splice register values into the context of the exception handler to be invoked.[6] It would be in principle possible to implement this interface in DIVINE system space: each thread would need two special thread-local variables to hold these values, and the `landingpad` instruction would simply read those values and copy them into appropriate LLVM registers. The downside is extra space overhead – 16 bytes per thread, allocated even if no exceptions are currently active.[7]

[6] This is clearly implemented in a platform-specific fashion. If the registers are always saved on the stack, their stack images will be rewritten. If they are clobber-type registers, they can be written to directly and the unwinder will take care not to clobber them before transferring control to the selected exception handler. Other options may be available depending on platform.

[7] Those 16 bytes could be compressed away in most cases to a single bit, at expense of code complexity. However, system-space complexity is very costly, and complexity involved in addressing the state vector even more so.

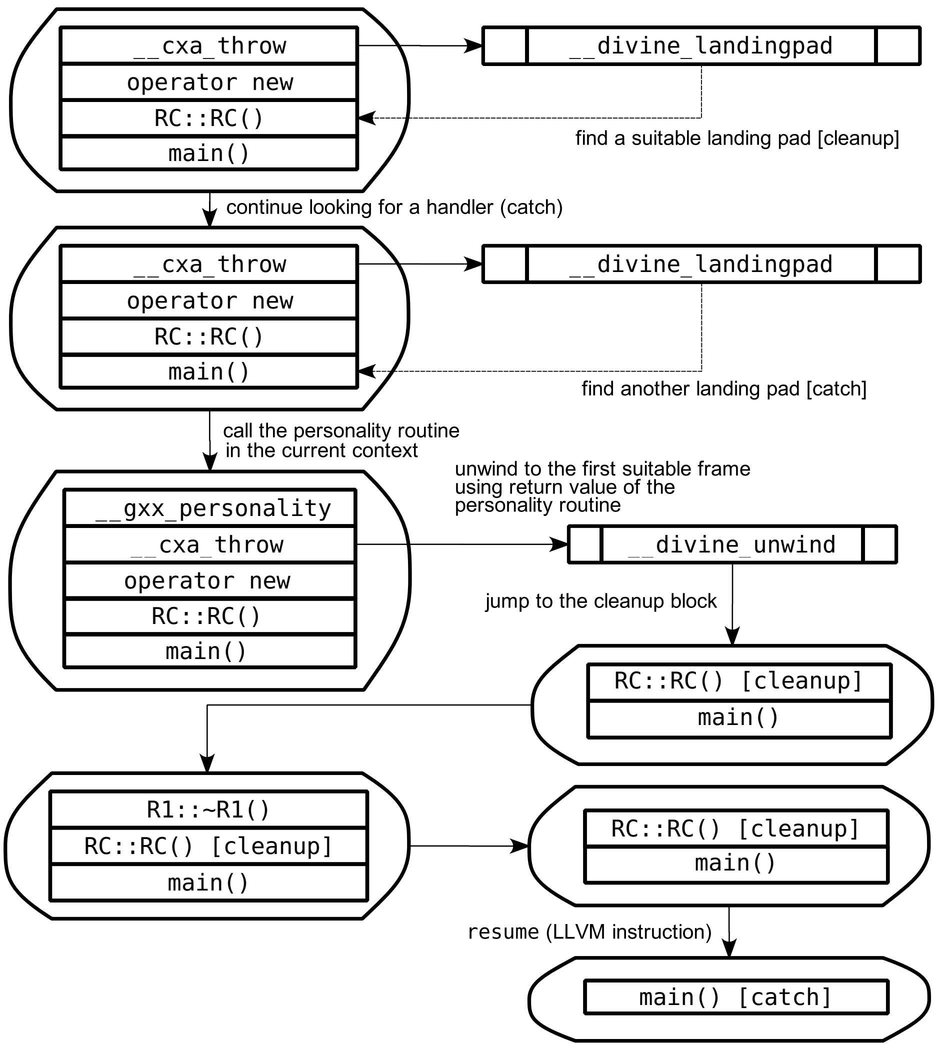
__cxa_throw
operator new
RC::RC()
main()
__divine_landingpad
find a suitable landing pad [cleanup]
continue looking for a handler (catch)
__cxa_throw
operator new
RC::RC()
main()
__divine_landingpad
find another landing pad [catch]
call the personality routine in the current context
unwind to the first suitable frame using return value of the personality routine
__gxx_personality
__cxa_throw
operator new
RC::RC()
main()
__divine_unwind
jump to the cleanup block
RC::RC() [cleanup]
main()
R1::~R1()
RC::RC() [cleanup]
main()
RC::RC() [cleanup]
main()
resume (LLVM instruction)
main() [catch]

Figure 3: Example of an exception-handling process as it happens in the DIVINE runtime (see Figure 2 for the source code). The situation at the top of the flowchart corresponds to an out-of-memory condition in the program. Constructor of class `RC` was trying to obtain dynamic memory (using operator `new`), but the allocation request has failed. As a result, operator `new` is throwing an exception – the `throw` statement in the C++ source code of the implementation is translated to a `__cxa_throw` call, which uses `__cxa_throw_divine` to unwind the stack. The unwinder first uses `__divine_landingpad` to find an exception handler (which it finds in the call frame of the `main()` function, and any intervening cleanup handlers (there is one in the `RC` constructor itself). The unwinder proceeds to call the personality routine to obtain a handler switch value and passes the result to `__divine_unwind`, along with the address of the first cleanup handler. `__divine_unwind` removes stack frames up to the cleanup handler, which takes control and calls a destructor of the locally constructed `R1` instance. Finally, when done, the cleanup handler invokes the `resume` instruction which continues the propagation up the stack, to the exception handler (the catch block in `main()`).

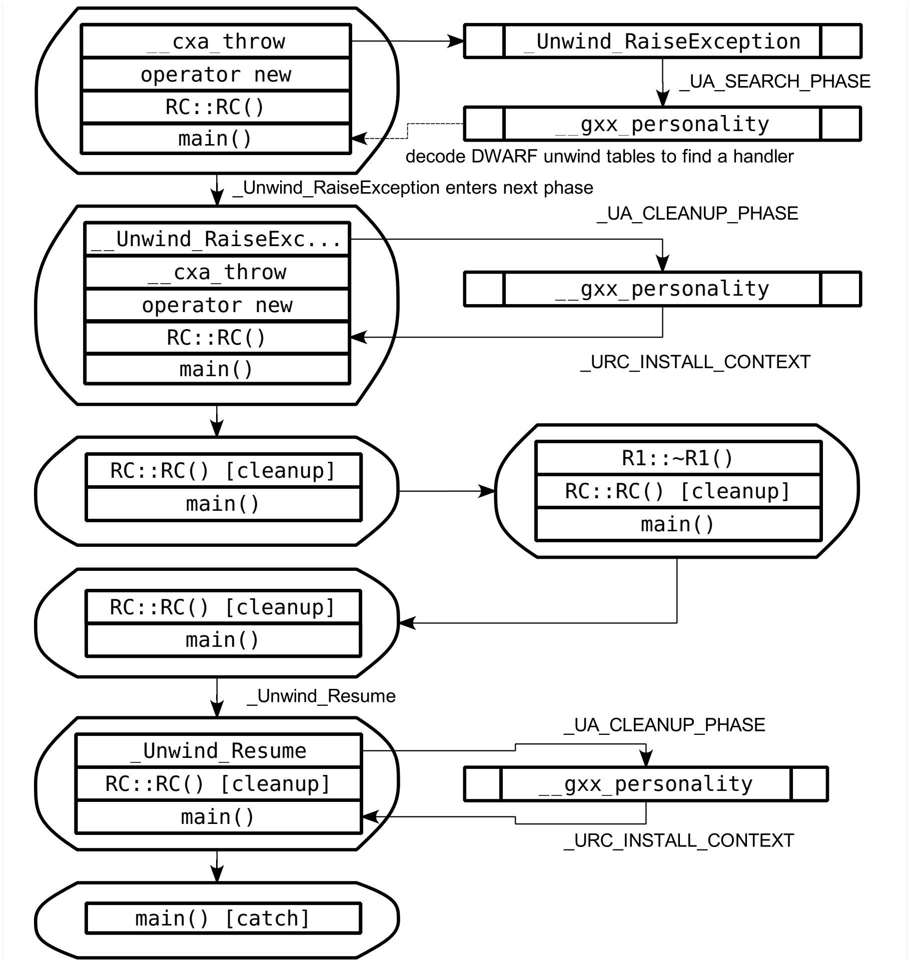

Figure 4: Example of an exception-handling process as it happens in the standard `libc++abi` process on 64-bit Linux (see Figure 2 for the source code and Figure 3 for comparison with DIVINE). The situation at the top of the flowchart corresponds to an out-of-memory condition in the program – as a result, operator `new` is throwing an exception – the `throw` statement in the C++ source code of the implementation is translated to a `__cxa_throw` call. The `__cxa_throw` implementation then calls into `libunwind` – the `_Unwind_RaiseException` function in particular. At this point, `libunwind` takes over control, looping over active stack frames. Each frame is examined by calling the personality routine with a `_UA_SEARCH_PHASE` flag, in the context of the `throw` statement. In this phase, an exception handler is identified, but the stack is not yet unwound. In the next phase, the stack is actually unwound, and again, each frame is examined by a call to the personality routine. If a cleanup handler or the selected exception handler is found, it is invoked by returning `_URC_INSTALL_CONTEXT` to `libunwind` (otherwise, `_URC_CONTINUE_UNWIND` indicates that unwinding should continue with the next frame). Cleanup handlers return control to `libunwind` by invoking `_Unwind_Resume`.

Another downside is that this limits flexibility: while the LLVM exception mechanism is made to play nice with libunwind, it is flexible enough, at least in theory, to admit another approach to stack unwinding. Using this approach would mean changing the DIVINE system space to accommodate a different landingpad return type.

While this API/ABI issue has reasonable solutions, there is a more important issue at play. While libunwind understands the platform-specific portions of unwind tables, it provides no support for parsing the language-specific chunks. This means that libc++abi code itself has ABI-specific knowledge of the unwind table layout, needed to extract the exception type info and switch values. All libunwind does here is provide a pointer to the lsda (language-specific data area) portion of the unwind table for a given stack frame. In order to support this libc++abi code in its literal form, DIVINE would have to synthesise DWARF-formatted[8] lsda areas from landingpad instructions. This is unpleasant, because it is a complex format designed for space efficiency, and the encoded tables are completely C++ specific, even specific to C++ on a particular platform. The only reasonable way to provide such tables would be to leverage pieces of the existing x86 (or x86-64) code generator to synthesise the lsda tables. LLVM, however, does not provide an interface to this functionality.

5.2 DIVINE-specific unwinding API

Both these issues in mind, we have chosen a different approach, which requires modifications to libc++abi, but can be implemented with just 2 new system-space builtins – one for querying metadata encoded in landingpad instructions, based on a stack frame reference (__divine_landingpad) and another for actually unwinding the stack (__divine_unwind).

This clearly requires some changes in libc++abi: one is the personality function, and the other is the actual __cxa_throw implementation: a call to this function is inserted by the C++ compiler at the site of a throw statement (along with some support code). While in the original libc++abi implementation, the personality function bears most of the burden (since libunwind does the stack search, calling out to the personality function as needed), this is reversed in the DIVINE implementation. Here, the personality function merely extracts the correct items from the exception header to pass on to the exception handler. The __cxa_throw implementation, on the other hand (and unlike in the libunwind version) unwinds the stack itself using __divine_landingpad. This builtin does not change anything, but provides the caller with landingpad metadata, using a simple integer indexing of stack frames. Negative indices start at the top of the stack, non-negative at the bottom. This makes it easy for the unwinder to walk through the stack one frame at a time, looking for an appropriate handler. When the handler is found, it can call the personality function (pointer to which is part of the landingpad metadata) and pass it to __divine_unwind along with the frame address it obtained from calling __divine_landingpad. The job of __divine_unwind is then simple enough: destroy all the frames above the one addressed and transfer control to the landing block associated with the active invoke instruction in the now-topmost stack frame. __divine_unwind also takes care of copying the value it obtained from its caller (in this case the return value of the personality

[8] DWARF is a companion format to encode debug and other metadata in ELF executable images. A backronym "Debugging With Attributed Record Format" has been invented for it.

function) into the result of the corresponding `landingpad` instruction.

The implementation of `__divine_landingpad` takes advantage of the implicit garbage collection done by DIVINE, as it allocates the metadata block on heap. Since the block is neither flagged as a result of an `alloca` instruction, nor as a result of `__divine_malloc`, it is transparently retained as long as necessary without being flagged by the interpreter as a memory leak.

5.3 `setjmp` and `longjmp`

The C functions `setjmp` and `longjmp` can be used for non-local transfer of control, in a way somewhat similar to C++ exceptions. In fact, some C programs use those two semi-standard functions to implement somewhat crude exception handling in C. The purpose of the `setjmp` function is to save enough of the machine state to allow non-local transfer of control to the point in program where `setjmp` was called. The `longjmp` partner then, using a context saved by the `setjmp` call, restores the corresponding machine state. The state is exactly the same as it was right after `setjmp` call returned for the first time, with one exception: the return value of the `setjmp` call is altered in its second return, to make it possible to detect whether the return was a "normal" return or a `longjmp` return.

Clearly, exception handling based on `setjmp`/`longjmp` cannot be "zero-cost" – state has to be explicitly saved at the start of every `try` block, and possibly before any resource acquisition. The latter problem can be side-stepped by maintaining a separate "resource" stack [Tur], but even then, entering `try` blocks is fairly expensive. Nevertheless, robust C programs may choose this style of exception handling, since the runtime overhead can be outweighed by the programming benefits – especially due to fewer and simpler error paths to write, maintain and test. Finally, there are other uses for `longjmp` in programs, besides exceptional situations.

While `longjmp` is not nearly as widely used as C++ exceptions are, the reasons for supporting this primitive are similar, even if somewhat weaker. Fortunately, the primitives we have designed for C++ exception handling can be easily re-used in implementing `setjmp` and `longjmp` – since `__divine_unwind` can just as easily stop at a `call` instruction as it can on an `invoke` instruction, we only need minor extensions to the `__divine_landingpad`/`__divine_unwind` mechanism. The main difference between exceptions and `longjmp` is how the control flow at the point of `setjmp` is handled. The DIVINE-specific implementation of `setjmp` needs to be able to find out the program counter value of its enclosing frame, corresponding to the call instruction. This can be done by slightly extending `__divine_landingpad`, to provide the program counter value for `call` instructions in the stack (this does not alter the semantics of `__divine_landingpad` for `invoke` instructions in any way).

Finally, `__divine_unwind` needs to be extended as well, to allow the caller to specify where to restart the execution in the target frame – since `longjmp` is not above the corresponding `setjmp` in the call stack, a successful `longjmp` needs to change the program counter in the target frame, in addition to unwinding. Luckily, this is fairly easy, since the `__divine_unwind` caller can specify the program counter corresponding to the callsite to unwind to. For normal C++ exceptions, the caller just puts in a 0, meaning no program counter adjustment (i.e. the semantics stay exactly the same) and `longjmp` passes in the program counter value obtained from a `__divine_landingpad` call done by the `setjmp` function.

6 Case Studies

Besides the simple fact of making model checking possible on a substantially wider class of programs, exceptions themselves are an interesting subject for model checkers: error paths are notoriously hard to test. With a model checker, however, it is easy to insert non-deterministic failures and check that the program behaves sensibly under all sorts of error conditions. Resource leaks are among the most common errors encountered in error paths, which makes the problem even harder to debug – resource leaks, especially memory leaks, require special tools to diagnose in a test, such as valgrind.

Since DIVINE can already diagnose memory leaks in LLVM inputs, checking error paths involving exceptions becomes a fairly easy task. However, error paths can contain more serious errors as well – especially in multi-threaded programs, where threads are not isolated from the effects of other threads failing to handle an exception, and the entire program may crash. Among the first issues that we have found using our new exception support in DIVINE is such a crash, in `std::thread` implementation in `libc++`, under out-of memory conditions.[9] When a new thread is created using this standard C++ interface, most of its state is allocated in the newly-created thread, before user code is executed. Since this allocation can fail with an exception, and the `libc++` implementation fails to install an exception handler in the context of this newly-created thread, the exception cannot be caught. In such cases, the C++ standard document requires the runtime library to call an "unexpected exception handler", which, unless overriden by the user, terminates the application.

In order to fix this problem, we have moved the memory allocation code into the calling thread. To avoid synchronisation problems and possible resource leaks, this happens before the new thread is created – the calling thread allocates all the dynamic state for the new thread and passes it down as a parameter. This way, any exceptions related to resource exhaustion happen in the calling thread, in a context where users can control the scope and propagation of exceptions by wrapping the call to the thread constructor in a suitable `catch` block.

7 Conclusions

We have shown how to extend an explicit-state software model checker based on LLVM with support for exception handling, with focus on C++ exceptions. To ensure the viability of the approach described in the paper, we have created an implementation as part of the DIVINE model checker. Additionally, we have used this implementation to verify properties of C++ programs that make use of exceptions (either directly or via the standard library) and in the process found an exception-related bug in the `libc++` implementation of `std::thread`.

To the best of our knowledge, this makes DIVINE the first model checker to be able to verify C++ code with exceptions[10]. The main contributions of the paper are twofold: first, the description of C++ and LLVM exception handling mechanisms in the context of model checking, and

[9] The proposed patch that fixes the problem can be found in http://llvm.org/bugs/show_bug.cgi?id=15638 and the relevant source code in the file `libc++/std/thread`.

[10] According to [RFS+13], the ESBMC++ tool also supports C++ exceptions. Unfortunately, it appears that this support is so far largely theoretical: the current version (1.23) produces a spurious counterexample on a simple test case, taking a path through the code which disregards the fact that an exception has been raised.

second the implementation derived from it: all relevant source code is freely available as part of the current DIVINE distribution.

Bibliography

[BBH$^+$13] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*. LNCS 8044, pp. 863–868. Springer, 2013.

[Bey14] D. Beyer. Status Report on Software Verification - (Competition Summary SV-COMP 2014). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*. LNCS 8413, pp. 373–388. Springer, 2014.

[CDE08] C. Cadar, D. Dunbar, D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2008)*. Pp. 209–224. USENIX Association, 2008.

[DWA10] DWARF Debugging Information Format Committee. *DWARF debugging information format version 4*. 2010.
http://dwarfstd.org/

[ELPP12] S. Evangelista, A. Laarman, L. Petrucci, J. van de Pol. Improved Multi-Core Nested Depth-First Search. In *Automated Technology for Verification and Analysis (ATVA 2012)*. LNCS 7561, pp. 269–283. Springer, 2012.

[LA04] C. Lattner, V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*. Palo Alto, California, Mar 2004.

[MFS12] F. Merz, S. Falke, C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*. VSTTE'12, pp. 146–161. Springer-Verlag, 2012.

[RFS$^+$13] M. Ramalho, M. Freitas, F. Sousa, H. Marques, L. Cordeiro, B. Fischer. SMT-Based Bounded Model Checking of C++ Programs. In *Proceedings of the 20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems*. ECBS '13, pp. 147–156. IEEE Computer Society, Washington, DC, USA, 2013.

[Tur] D. Turner. Robust Design Techniques for C Programs.
http://freetype.sourceforge.net/david/reliable-c.html

[VW86] M. Vardi, P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *IEEE Symposium on Logic in Computer Science*. Pp. 322–331. Computer Society Press, 1986.