# Borealis Bounded Model Checker: The Coming of Age Story

**Marat Akhin, Mikhail Belyaev, and Vladimir Itsykson**

**Abstract** Our research group has been developing a bounded model checker called Borealis for almost 4 years now, and it has been mostly a research prototype with all that it entails. A lot of different ideas have been tested in Borealis, and this chapter draws a bottom line for most of them. We believe this chapter would be of interest to other researchers as a brief introduction to the topic of bounded model checking, and to us as a cornerstone on which to build our future work on making Borealis into a tool.

## 1 Introduction

Development of any program analysis tool is a very interesting journey, as it compels one to delve into many different areas, starting from logic and formal semantics to language design and compiler construction. But it is also a very difficult journey with a lot of [undecidable] problems you need to solve to make your tool work.

This chapter is the *summa experitur* of our journey in the development of a bounded model checking tool nicknamed Borealis.[1,2] We talk about the problems most often encountered in bounded model checking and how we decided to tackle them in Borealis. We also attempt to explain the reasonings about our decisions, to share not only the *results* but also the *process*. In a sense, this chapter is our personal cornerstone in Borealis' development, which summarizes what has already been done and records our plans for the future.

The rest of the chapter is organized as follows. Section 2 introduces the basics of bounded model checking in general and its implementation in Borealis. Our takes on the problems of loop and interprocedural analyses are described in Sects. 3 and

[1]https://bitbucket.org/vorpal-research/borealis.

[2]https://hub.docker.com/r/vorpal/borealis-standalone/.

M. Akhin (✉) • M. Belyaev • V. Itsykson
Peter the Great St. Petersburg Polytechnic University, Saint Petersburg, Russia
e-mail: akhin@kspt.icc.spbstu.ru; belyaev@kspt.icc.spbstu.ru; vlad@icc.spbstu.ru

M. Mazzara, B. Meyer (eds.), *Present and Ulterior Software Engineering*,
https://doi.org/10.1007/978-3-319-67425-4_8

4, respectively. In Sect. 5 we present our approach to the precision/recall problem, which is based on a semi-symbolic program execution. We talk about the practical evaluation of Borealis in Sect. 6; Sect. 7 wraps up with some discussion of our future work.

## 2 The Basics of Borealis Bounded Model Checker

As simple and trivial as it may seem to a more experienced reader, we still believe we should begin our chronicles of Borealis' research and development from the very basics. Therefore, in this section we outline the inner workings of Borealis, both to set a context for our work and to make a solid foundation for the rest of this chapter.

The first step in developing any bounded model checking (BMC) tool for a given programming language is to decide what "M" stands for, i.e., which program model you are going to use. In principle, one has three possible options to choose from:

1. OSS compilers (e.g., GCC or Clang)
2. Proprietary compilers (e.g., EDG)
3. Custom-made parsers/compilers

While the last option looks very attractive, as handcrafted tools can be tailor-fit to do exactly what is needed, creating even a parser (let alone a fully functional compiler) for a language as complex as C++ is no small feat. Paying for a proprietary compiler framework is also not an option, if we are talking about small-scale R & D. Therefore, one is left with using OSS compilers to develop her BMC tools.

There are a lot of tools for a lot of programming languages. We targeted C and chose to go with LLVM framework,[3] as it has some unique advantages compared to other alternatives. First, it was created with program analysis in mind right from the start. LLVM intermediate representation (IR) is a highly regular, typed language in static single assignment (SSA) form which makes developing analyses easy. Second, it provides a large number of analyses and optimizations, ranging from alias analyses to loop unrollings, out of the box, allowing one to focus on their problem domain and not on reimplementing collateral facilities. Third, it is designed for extensibility and composability, making it very easy to build tools with it.

As more and more software projects are switching to Clang/LLVM as their main toolchain, using LLVM for program analysis has an additional advantage of being *compiler aware*—tools built on LLVM platform are analyzing the code exactly as it is seen by the actual compiler, leaving almost no space for possible misinterpretations between the compiler and the analyzer (which are easily possible with C/C++). Of course, one might argue that the LLVM framework itself might work with the code erroneously, e.g., w.r.t. undefined behavior [30, 33], which would

[3]http://llvm.org/.

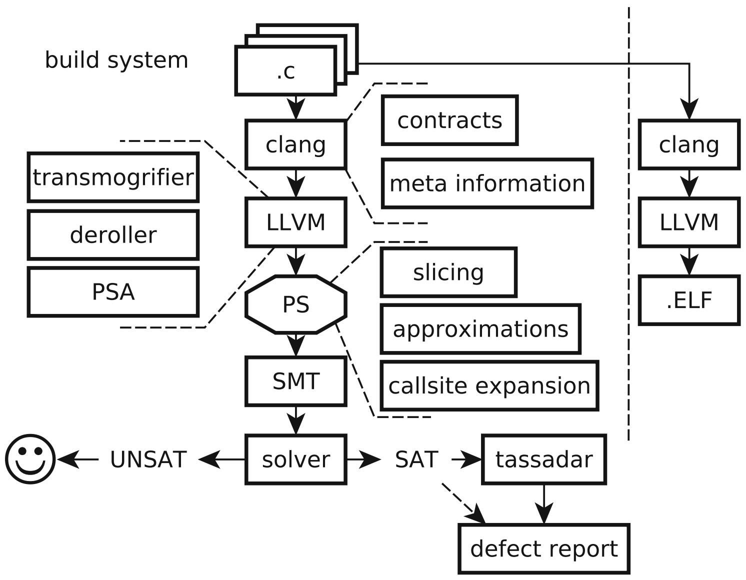


**Fig. 1** High-level scheme of Borealis bounded model checker

lead to analysis tools also interpreting the code in the wrong way, but this problem lies outside the scope of this chapter.

The high-level scheme of Borealis is shown in Fig. 1. Ad initium, let us explain the main Borealis components in more detail.

## *2.1 Low-Level Virtual Machine Infrastructure*

Borealis is built upon the standard foundation of almost every LLVM-based tool: LLVM passes. A single pass is a function that takes some LLVM IR module, either analyzes or transforms it, and returns the resulting LLVM IR module. This allows one to easily combine several passes in a processing pipeline according to her task.

As with any BMC tool, Borealis requires the code to be fully unrolled,[4] i.e., no loops are to be present in the resulting LLVM IR. To support this, we implemented a `loop-deroll` pass which is a more aggressive version of the standard LLVM `loop-unroll` pass. It also implements a novel loop unrolling technique called *loop backstabbing* which is covered in more detail in Sect. 3.2. This pass, together with a number of built-in LLVM passes (e.g., *aggressive dead code elimination*

[4]That is, bounded.

or *promote memory to register*), is used to make the program's LLVM IR suitable for BMC.

There are a number of other auxiliary passes that are responsible for such tasks as metadata bookkeeping (managing the flow of code metadata from Clang through LLVM to Borealis), annotation processing (see Sect. 2.3), and function decomposing (see Sect. 4.2). After all these passes are done, the LLVM module follows the next requirements:

- All functions are fully unrolled (their control flow graphs are directed acyclic graphs).
- External function calls are replaced with their decomposed representation.
- Every variable is tagged with its detailed metadata (e.g., C type, original name, source code location).
- Source-level annotations are embedded in LLVM IR via special intrinsics.

This resulting LLVM IR needs to be checked for possible bugs. To do that, we employ the power of modern satisfiability modulo theories (SMT) solvers [7].

## 2.2 *Satisfiability Modulo Theories*

The idea behind utilizing SMT for BMC is a very simple one—transform the unrolled program to an SMT formula and check if it is satisfiable w.r.t. some safeness condition [12]. If the condition is violated (formula is satisfiable), the original program contains a possible bug, otherwise it is correct. Checking is done using one of the available SMT solvers, e.g., Z3 [23] or MathSAT [14].

To make supporting different solvers easier, Borealis adds another level of indirection in the form of intermediate *predicate state* API (PS API). A simplified PS definition is shown in Fig. 2; a PS for a given LLVM IR instruction corresponds to an SMT formula equivalent to all the possible program states at that LLVM IR instruction. Instructions is mapped to *predicates*, which operate on *terms* (our equivalent of LLVM IR values). Predicate states are easily composable, retain only the information needed for LLVM-to-SMT translation, and can be efficiently (de)serialized (a feature required for incremental-style analyses).

To do the actual PS-to-SMT translation, one needs to define the transformation rules, and this is where everything gets interesting. Most terms and predicates are self-explanatory: `TernaryTerm` selects one of the two values depending on the third, `GepTerm` corresponds to LLVM `GetElementPtr` instruction, etc.; program values are represented as SMT bit vectors of their respective size. However, there are some terms which work with memory, and memory cannot be easily encoded in SMT.

We decided to use the approach similar to LLBMC [28], where memory is represented as a theoretical array with `read` (`LoadTerm`) and `write` (`StorePredicate`) operations [13]. This allows for much flexibility of if

```
⟨PredicateState⟩ ::= PredicateStateChain head:⟨PredicateState⟩ tail:⟨PredicateState⟩
   | PredicateStateChoice choices:⟨ListOfPredicateStates⟩
   | BasicPredicateState data:⟨ListOfPredicates⟩

⟨Predicate⟩ ::= AllocaPredicate lhv:⟨Term⟩ numElems:⟨Term⟩ origNumElems:⟨Term⟩
   | DefaultSwitchCasePredicate cond:⟨Term⟩ cases:⟨ListOfTerms⟩
   | EqualityPredicate lhv:⟨Term⟩ rhv:⟨Term⟩
   | GlobalsPredicate globals:⟨ListOfTerms⟩
   | InequalityPredicate lhv:⟨Term⟩ rhv:⟨Term⟩
   | MallocPredicate lhv:⟨Term⟩ numElems:⟨Term⟩ origNumElems:⟨Term⟩
   | SeqDataPredicate base:⟨Term⟩ data:⟨ListOfTerms⟩
   | SeqDataZeroPredicate base:⟨Term⟩ size: UInt32
   | StorePredicate ptr:⟨Term⟩ value:⟨Term⟩
   | WriteBoundPredicate ptr:⟨Term⟩ boundValue:⟨Term⟩
   | WritePropertyPredicate propName:⟨Term⟩ ptr:⟨Term⟩ propValue:⟨Term⟩

⟨Term⟩ ::= ArgumentTerm idx: UInt32 kind:⟨ArgumentKind⟩
   | ArgumentCountTerm
   | AxiomTerm term:⟨Term⟩ axiom:⟨Term⟩
   | BinaryTerm op:⟨BinaryOp⟩ lhv:⟨Term⟩ rhv:⟨Term⟩
   | BoundTerm term:⟨Term⟩
   | CastTerm term:⟨Term⟩ signExtend:Bool
   | CmpTerm op:⟨CmpOp⟩ lhv:⟨Term⟩ rhv:⟨Term⟩
   | ConstTerm
   | FreeVarTerm
   | GepTerm base:⟨Term⟩ shifts:⟨ListOfTerms⟩ triviallyInbounds:Bool
   | LoadTerm ptr:⟨Term⟩
   | ReadPropertyTerm propName:⟨Term⟩ ptr:⟨Term⟩
   | ReturnPtrTerm funcName:String
   | ReturnValueTerm funcName:String
   | SignTerm value:⟨Term⟩
   | TernaryTerm cond:⟨Term⟩ tru:⟨Term⟩ fls:⟨Term⟩
   | UnaryTerm op:⟨UnaryOp⟩ value:⟨Term⟩
   | ValueTerm global:Bool
   | VarArgumentTerm index: UInt32
   | ...

⟨ListOfPredicateStates⟩ ::= ⟨PredicateState⟩ ⟨ListOfPredicateStates⟩ | ⟨empty⟩

⟨ListOfPredicates⟩ ::= ⟨Predicate⟩ ⟨ListOfPredicates⟩ | ⟨empty⟩

⟨ListOfTerms⟩ ::= ⟨Term⟩ ⟨ListOfTerms⟩ | ⟨empty⟩

⟨ArgumentKind⟩ ::= ANY | STRING

⟨CmpOp⟩ ::= EQ | NEQ | GT | ...

⟨BinaryOp⟩ ::= ADD | SUB | MUL | ...

⟨UnaryOp⟩ ::= NEG | NOT | BINARY_NOT
```

**Fig. 2** Predicate state definition

and how we interpret bit-wise operations—we can use byte- or element-level memory representation or any combination in between. After tests we decided to use element-level representation (every program value is mapped to a single memory element) by default, as it is efficient and precise enough for most practical BMC purposes [2].

Besides the main memory, which represents the program heap, Borealis uses auxiliary *property* memories. These named memories store additional values linked to program values, e.g., every heap-allocated pointer has its bound stored in $\langle bounds \rangle$ memory. This allows us to save any kind of relevant information without the need to change Borealis' internals. For example, if one needs to work with resources and resource states, these can be stored in $\langle resources \rangle$ memory and checked for bugs when needed.

The main alternative to array-based memory modeling in SMT is predicate abstraction [21, 24]. In case of predicate abstraction, memory operations are lifted to special predicate variables, which capture only the most important slices of the memory model. If these slices happen to be inadequate for a given bug, they may be refined using, for example, counterexample-guided abstraction refinement (CEGAR [17]).

## 2.3 Bug Detection

After predicate state corresponding to a given program fragment is converted to an SMT formula, it needs to be checked for correctness. We employ the traditional approach to checking if condition $Q$ always holds for formula $B$ by asking if $B \wedge \neg Q$ is satisfiable. If it is unsatisfiable, $B \wedge Q$ is always true and the program is correct; if it is satisfiable, the program contains a bug and the resulting SMT model summarizes the fault-inducing counterexample.

Borealis has a built-in support for the following bug types:

- Null-pointer dereferences
- Buffer overflows
- Uses of undefined values
- Code assert violations
- Function contract violations

The safety conditions for these bugs are pretty straightforward, so we leave them outside the scope of this chapter.

To specify custom safety conditions (i.e., function pre- and post-conditions), Borealis uses two flavors of annotations. The first one, called $\natural$ACSL,[5] is heavily inspired by ANSI/ISO C Specification Language (ACSL) [8]; the specifications are given as annotations in source code comments [2] which allow to describe

[5]Read as "natural-ACSL".

function pre- (`@requires`) and post-conditions (`@ensures`), arbitrary assumptions (`@assume`), and custom checks (`@assert`). The other one is more in line with SV-COMP-styled checks [11], when conditions are embedded in the source code via calls to special intrinsic functions. While the latter mode is enough for many practical purposes (and is a de facto standard in software verification), the former allows for much extensibility and expressibility of complex safety conditions.

Now that we are done with the basics, we can go on. Specifically, we are going to talk about our takes on two major problems in software verification: analysis of loops and interprocedural analysis.

## 3 Program Loop Analysis

As we already mentioned in Sect. 2, bounded model checking expects the code to be fully unrolled in order to be converted to an SMT formula (as BMC is, by definition, "bounded"). Unfortunately, this problem is undecidable in general, as it requires one to solve the halting problem. In the following sections we describe different approaches, which can be used to implement this unrolling, and the motivation behind the hybrid approach we use in our tool.

### *3.1 The Unroll-and-Bound Technique*

The easiest (and most common) way of dealing with program loops is unrolling each loop in the program by a constant unrolling factor. This technique is implemented in most bounded model checkers [4, 18, 19, 28] due to its simplicity, but is unsound in general. Additionally, some classes of software bugs (namely, buffer overflows) usually do not trigger on the first iterations of the loop, meaning that this technique performs poorly on these bugs.

The classic approach to constant loop unrolling is as follows: First, some heuristic-based method is used to find out whether a loop has a constant iteration count. If it does, the loop is fully unrolled up to the last iteration, and the analysis is sound by construction. If it does not, the loop is unrolled to some arbitrary constant value, and the analysis attempts to prove the program correct. It is easy to show that the value of this constant is insignificant in general (as long as it is greater than one); as for any possible constant value $N$ sufficient to analyze any given program, a new program can be crafted requiring an unroll factor of at least $(N + 1)$.

The original BMC paper [12] describes an approach that aims to solve this problem by unrolling the original program up to a point where further unrollings do not change the analysis results. This approach does minify the unsoundness of the constant loop unrolling in general, but requires re-running the analysis with a different unrolling factor for every loop, thus introducing a multiplicative increase in execution time.

## 3.2 Loop Backstabbing

The unroll-and-bound technique described previously is a very simple and universal, but generally unsound, way of dealing with loops, as every iteration is different if we take the time-centric view. However, a large number of loops in typical C programs are pretty standard in their form and function (despite not having a constant unroll factor) [25]. There are techniques for reasoning about values in these kinds of loops as a single abstract entity rather than a sequence of values in time. An example of such technique is *chains of recurrences* (CR) abstraction [6, 32], which we will now describe in more detail.

### 3.2.1 Chains of Recurrences

A chains of recurrences (CR) expression is a special kind of symbolic expression that has the following form:

$$\Phi = \{\phi_0, \odot_0, \phi_1, \odot_1, \ldots, \phi_{k-1}, \odot_k, \phi_k\}_i$$

which is a short form of

$$\Phi = \{\phi_0, \odot_0, \{\phi_1, \odot_1, \ldots, \{\phi_{k-1}, \odot_k, \phi_k\}_i \ldots\}_i\}_i$$

where each tuple

$$f_j(i) = \{\phi_j, \odot_{j+1}, f_{j+1}\}_i$$

(also called *basic recurrence* (BR)) corresponds to a recursive formula

$$f_j(i) = \begin{cases} \phi_j & \text{if } i = 0 \\ f_j(i-1) \odot_{j+1} f_{j+1}(i-1) & \text{if } i > 0 \end{cases}$$

Each operation $\odot_k$ may be either $+$ (sum recurrence) or $*$ (product recurrence). In principle, BRs can be extended to handle other operations, but they are either too complex to operate on (e.g., exponents) or reducible to sum/product recurrences (e.g., subtraction and division). Each expression $\phi_k$ may be bound to any symbolic expression (possibly including other CR).

CR framework can, in general, represent a wide range of recurrently defined expressions with a seed (starting value), e.g., different kinds of loop variants such as loop-dependent expressions or induction variables. Of course, some loop variants cannot be represented as CR, most notably, values which are based on several other values from previous iterations.

After being constructed, CR may be "unrolled" to an arbitrary iteration $k$, where $k$ is a symbolic expression, and the unrolled version can be converted back to the nonrecurrent form for a large subset of all possible CR, thus providing the means to symbolically evaluate an arbitrary recurrent symbolic expression at any iteration, even if the iteration number is symbolic as well. The detailed algorithm can be found in paper [32].

CR are widely employed as a code analysis technique in compilers to reason about value evolution. They improve the *scalar evolution* analysis in both LLVM [26] and GCC [10] frameworks and are the foundation of many modern code vectorization techniques.

### 3.2.2 Using Chains of Recurrences for Loop Analysis

If we consider a program with no side effects or memory accesses, a set of CR expressions for all inductive variables together with a set of CR-dependent expressions for all noninductive variables summarizes any given loop. On top of this, if all CR expressions can be converted back to their nonrecurrent forms, the whole loop can be turned into a universally quantified formula over the number of iterations $i$. Such a formula can then be used by bounded model checking as is, to represent the body of the loop.

There are, however, several problems with this approach. First, real programs do contain memory access and side effects. Second, one generally tries to avoid producing quantified formulas, as they are hard to reason about and may slow down the SMT solver due to the general undecidability of the quantifier elimination.

In order to avoid these problems, we make the following assumption. As the process of finding a safety property or contract violation (see Sect. 2) uses the formula for the whole function and makes the solver infer the values of all the variables in this formula, we assume only one iteration of a loop is needed for the analysis. That is, if a single "in-the-middle" iteration triggers the bug, the solver should be able to find this exact iteration and produce a subsequent counterexample. This assumption does not hold, however, if a violation requires something happening on two or more iterations of the same loop, but we believe this situation happens in a much smaller number of cases. Even more so, we believe the more iterations are needed to trigger a violation, the less likely this situation is in practice. Despite being without any theoretical proof, these assumptions show good results in practice [1].

Under these assumptions, we can replace a *universally* quantified formula $\forall\, i.\Psi(i)$, which summarizes a loop, with a set of *existentially* quantified formulae:

$$\exists(i \neq j \neq \cdots \neq k).\Psi_0(i) \wedge \Psi_1(j) \wedge \cdots \wedge \Psi_N(k)$$

It might seem we did not solve any of the problems we had before, but we actually did. These existentially quantified formulae not only represent the loop w.r.t. possible side effects but also allow to trivially skolemize the quantifiers, while producing a counterexample for a possible violation, thus reducing the formulae to the following:

$$(i \neq j \neq \cdots \neq k) \wedge \Psi_0(i) \wedge \Psi_1(j) \wedge \cdots \wedge \Psi_N(k) \text{ where } (i, j, \ldots, k) \text{ are fresh variables}$$

Another thing to consider is `continue`/`break` instructions in a loop—in the proposed CR loop representation, these instructions map to additional predicates over the induction variables $(i, j, \ldots, k)$. In order to add these predicates, we actually need to restructure the program, so that we do not introduce one-too-many iterations when doing loop unrolling.

Surprisingly enough, this technique can be easily adapted to handle infinite loops. Infinite loops happen to not have break instructions; therefore, we can just skip their addition to the resulting formulae. The rest of the approach works in the same way as for finite loops.

### 3.2.3 Loop Backstabbing Implementation

The actual implementation of loop backstabbing in Borealis applies CR transformation to a loop and extracts the first $K$ (as in classic unroll-and-bound), an arbitrary iteration in the middle, and the last iteration of a loop. The biggest advantage over the classic unroll-and-bound is that these auxiliary iterations are bound by symbolic variables, and the solver may consider *any* iteration of the loop which leads to a possible bug. If the number of iterations of a particular loop is constant, and this constant is less than a certain configurable threshold, we fall back to the unroll-and-bound technique, using built-in LLVM tools. All the loops are analyzed and unrolled in the bottom-up order to avoid potential cross-loop-induction problems.

We use the built-in LLVM scalar evolution analysis, as it is a production-ready implementation easily available from within the compiler. However, while evaluating this approach, we found its CR API to be quite limited, not being able to handle exponential CR and too closely tied to the LLVM loop analysis infrastructure. In our future work we plan to explore the options of storing CR directly in the program representation and using them to also handle some cases of recursion, as well as more complex loops, which all require a full-blown standalone CR implementation.

We refer to this technique as *loop backstabbing*, as it tries to avoid the "start-from-the-beginning" approach of the classic unrolling (hence, "back-") and finds the exact erroneous spot of the loop execution (hence, "-stabbing"). Further details and evaluation of loop backstabbing can be found in [1].

## 4 Interprocedural Analysis

Another serious problem for any kind of program analysis is dealing with function calls, as any function call may have an unknown influence on the program state. There are quite a few possible ways of dealing with the problem of interprocedural analysis, and we explored most of them at some point of Borealis' research and development.

### 4.1 Inlining

Function inlining has always been the traditional "go-to" approach to interprocedural analysis in the BMC setting. It consists of full function body inlining at every interesting call site. While this approach fits BMC perfectly and works quite well for some programs, it has a number of downsides. The main disadvantage of inlining is that it significantly increases the state space size (in the worst case, exponentially in the depth of nested function calls), which, in turn, increases the resource requirements. It also poorly supports recursive function calls, as they require an iterative procedure similar to classic iterative loop unrolling (see Sect. 3.1).

We support this approach to interprocedural analysis by implementing a special LLVM `force-inline` pass, which, unlike the standard `inline` pass, always forces inlining. The rest of the processing pipeline is completely agnostic to the [absence of] inlining, which allows us to toggle inlining simply by enabling or disabling the `force-inline` pass.

All other approaches to interprocedural analysis reduce down to function approximation—a succinct representation of the function body that retains properties interesting to the program analysis. One option is to leave the function approximation to the user, who then provides function annotations; another is based on automatic approximation via logic inference. These options are described in more detail below.

### 4.2 Decomposition

The most simple way to approximate function calls (and the only way to deal with external libraries in general) is by using some function description for each external function. Annotations for functions internal to the project are provided using the already discussed ♮ACSL language (see Sect. 2.3) as structured comments in their source code. External functions need to be processed separately, as their source code cannot be easily edited. We decided to decompose every external function call into a number of basic operations over its arguments and result value, which capture its influence on the caller.

This decomposition is specified using a simple JSON-based description language. Each function description contains the ♮ACSL function contract (expressed over function parameters and result value) and its *access pattern*. The access pattern captures the information about how this function accesses memory. For each function parameter, we have an *access operation* $AOp$, which can have a value of `Read`, `Write`, `ReadWrite`, or `None`. For every parameter that is read from, an LLVM `load` instruction is generated; for parameters written to, we create `store` instructions. More complex access patterns can be expressed using `None` as an access operation and an appropriate ♮ACSL contract via `@assigns` and `@ensures`.

We use two separate mechanisms (access patterns and contract annotations) because, in many simple cases, the pattern-generated instructions can be optimized away, making the resulting code much easier to analyze; contract annotations, being separate from the code, cannot be optimized as easily.

## 4.3 Interpolation

Unfortunately, user-provided annotations are usually few and far between, and one has to automatically approximate a given function for the purposes of interprocedural analysis. Craig interpolation has been one of the most used approaches in BMC for quite some time now [20, 27]. It is based on the following premise: for any unsatisfiable pair of formulae $(B, Q)$, there exists a formula $I$ (called Craig interpolant) that satisfies the following conditions:

- $B \rightarrow I$.
- $(I, Q)$ is unsatisfiable.
- $I$ contains only uninterpreted symbols common to both $B$ and $Q$.

Formula $I$ can be viewed as an overapproximation of $B$, which means it is a function overapproximation if we take function body as $B$, its safeness property as $Q$, and our pair of formulae as $(B, \neg Q)$.

The main problem of applying Craig interpolation in practice is that an interpolant exists only for a pair of *unsatisfiable* formulae, i.e., if there is no bug in the program. If a bug is present, one might fall back to function inlining (as proposed in [31]) and skip function approximation completely. We propose another approach that tries to apply Craig interpolation to a satisfiable pair of formulae $(B, \neg Q)$ by using random model sampling [3].

The idea behind random model sampling is to strengthen the satisfiable formula $B \wedge \neg Q$ with additional premises $M$, such that $B \wedge \neg Q \wedge M$ becomes unsatisfiable. The outline of the random model sampling algorithm is presented in Fig. 3.

We iteratively probe possible satisfying models of $B \wedge Q$, restrict them to contain only function arguments, and retain only those that falsify the original formula. Craig interpolation is applied afterward to $B \wedge \neg Q \wedge FA$ to obtain the result.

```
if $B \wedge \neg Q$ is $UNSAT$ then
    Use regular Craig interpolation with $B \wedge \neg Q$
else
    $FA$ = false
    repeat
        $SA_i = SAT$ model for $B \wedge Q$                    $\triangleright$ excluding all previous models
        $FA_i = \{A \in SA_i | A$ includes a function argument$\}$
        if $B \wedge \neg Q \wedge FA_i$ is $SAT$ then
            continue
        end if
        $FA = FA \vee FA_i$
    until $FA$ has enough samples
    Use regular Craig interpolation with $B \wedge \neg Q \wedge FA$
end if
```

**Fig. 3** Random model sampling algorithm

To infer a function summary, we interpolate from function body $B$ to its interpolant $S$ over $\neg Q \wedge FA$. If $S \wedge \neg Q$ is unsatisfiable, we managed to extract an interesting function summary. If not, we could retry with different probes or apply another approach to interprocedural analysis. For additional details and evaluation results, see [3].

## 5 Improving Analysis Precision Using Semi-Symbolic Program Execution

As seen in the previous sections, before the program is analyzed, it undergoes quite a number of transformations and approximations, including (but not limited to) a simplified memory model, loop unrollings, and interprocedural approximations. All these simplifications often result in differences between the analysis results and the actual program behavior if run. In other words, the trade-off between analysis quality and performance leads to its precision and recall being less than 100%.

To be practically applicable, the analysis should have high precision, so that its results can be trusted by the users [5, 16]. To improve Borealis' precision without sacrificing performance, we augmented it via semi-symbolic program execution.

### *5.1 The Need to Improve Analysis Precision*

There are two basic measures of analysis quality: *precision* and *recall* [15]. They are dual in that one cannot achieve 100% precision and recall at the same time—any improvement to recall tends to raise the number of false positives (lowering precision), and any change which increases precision usually lowers the number of true positives (impacting recall).

The core idea of our approach is a very simple one: if one cannot lower the false positives rate by tweaking the analysis, one could try to filter out the false positives *after* the analysis has finished, thus having no impact on recall. We do this by executing the program IR in a checked and fully analysis-aware environment; in essence, this simulates a dynamic execution of the analyzed program, which (by definition) has 100% precision.

This semi-symbolic execution must satisfy the following properties to be applicable for our purposes:

- Avoid infinite or otherwise lengthy executions.
- Minimize resource consumption.
- Mitigate as many of analysis' inaccuracies as possible.
- Capture all interesting errors.

Let us explain how we achieve these properties in practice.

## 5.2 *Semi-Symbolic Code Execution in a Controlled Environment*

If we ignore the side effects and memory accesses, executing code in SSA form is pretty straightforward. We do need to explicitly model both the call stack and the instruction pointer register, but the rest can be implemented by storing a flat `Instruction -> Value` table. As no instruction is visited twice due to loop unrolling, each value in this table is effectively immutable.

Some values (e.g., `rand()` results) should be handled in an analysis-aware way, i.e., the values during the execution should be the same as were inferred by the analysis. This should also hold for things such as initial function arguments and global variables; otherwise, the execution would not conform to the analysis results. To support this, we treat these values as unknown or *symbolic*. Unlike classic symbolic execution, however, we try to get concrete values for as much of these symbols as possible using a special *arbiter*.

Arbiter is basically a function from symbols to values, which uses external information about the code. When checking BMC counterexamples, the arbiter provides the connection between the executor and the analysis results (i.e., values from the SMT satisfying assignment). However, the arbiter is independent from the analysis and could be extended upon if needed.

A very special case of symbolic values are pointers, which are modeled using special non-null values known both to the arbiter and the memory model. Dereferencing such a value queries the arbiter instead of the memory. As of now, we do not support writing values to symbolic pointers, which presents a problem when dealing with structures passed by pointers to top-level functions. This problem can be dealt with if one reconstructs the shape graph from the SMT counterexample; it is possible, but quite complex and not yet implemented in Borealis.

As code annotations are not compiled to LLVM IR, we also need to handle them separately in the executor. The contracts are represented as special intrinsics, so when the executor encounters such an instruction, it gets the values needed for the contract interpretation either from the arbiter or the current stack frame and evaluates the contract AST; as the contracts have no side effects, this step is pretty trivial.

## *5.3 Memory Modeling*

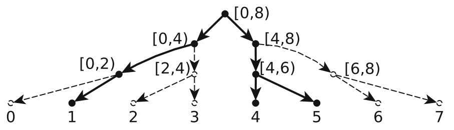
Modeling memory efficiently is a challenging task, which we tackle using *implicit segment trees*. A segment tree (ST) is a well-known data structure [22]; in essence, it is a balanced tree with leaves representing elements and inner nodes representing ranges, the root of the tree being the whole range, and every child node representing a range one-half of its parent's. This data structure can be used to efficiently perform a number of tasks, such as range queries with associative binary operation—changing a leaf value and recalculating the whole tree is $O(\log_2(N))$ for a range of size $N$.

The basic segment tree is not very efficient memory-wise, so we implemented a number of optimizations. First, the nodes of the tree are constructed lazily. Second, we reduce the height of the tree by storing a flat array of bytes at every node. Figure 4 shows these differences for our implicit segment tree.

The memory tree is structured as follows. Each node in the tree contains a tuple $\langle Q, S, F, Sz, D\rangle$. Memory status $Q$ (`Alloca`, `Malloc`, or `Unknown`) defines if this node is used as a root of an allocated memory region of size $Sz$. Memory state $S$ (`Memset`, `Uninitialized`, or `Unknown`) represents `memset`'ed, uninitialized, or undefined memory region, thus shortcutting reads for allocated-but-never-written-to regions. The value of $F$ is the `memset` value; the value of $D$ is the actual stored data.

For a given memory access, the address used is mapped to an index in the tree, which is then accessed in a top-down fashion, lazily creating intermediate nodes if needed. Read accesses may stop at a `memset` node or go to a leaf; write accesses always go to a leaf, changing the flags on nodes affected by write accordingly. Allocations do not create new nodes, as our tree is lazy, so we need to only set the corresponding memory status flags. A detailed memory access algorithm may be found in [9].

**Fig. 4** Implicit segment tree



### *5.4 Handling External Functions*

Another thing we need to execute actual C programs is to support external function calls, e.g., calls to the C standard library. Our executor uses a stub implementation of the C library and most common POSIX extensions. It does not simulate the actual side effects, but the values returned are sensible enough for most analysis purposes.

Functions directly operating with the memory are implemented on top of primitive `read`, `write`, `memset`, and `memchr` calls. `memcpy` is not a primitive operation at the moment and is implemented rather inefficiently, we plan to explore this problem in our future work. If one needs to provide additional external function implementations, there is no built-in support for this at the moment; however, one can use the other function approximation facilities provided by Borealis (see Sect. 4).

The executor (code named Tassadar) is implemented as a separate LLVM pass on top of Borealis' infrastructure, but is mostly independent from it. The actual execution engine is a derivative of the LLVM interpreter, with everything related to memory and contracts rewritten from scratch. For more details and evaluation, refer to [9].

## 6 Evaluation on Real-World Software

As mentioned before, most of small-scale evaluations for Borealis can be found in our previous papers [1–3, 9], where we focused more on if the sketched approaches actually improve anything. In this section we want to talk about our recent experiences with large-scale [attempts at] evaluation on industrial-sized projects.

There is a huge chasm between crafted programs traditionally used for software analyzer testings and real-world programs created to solve real-world problems. Our previous evaluations were done on NECLA and SV-COMP benchmarks, which have complex behavior, but are relatively small and written in a structured fashion. When you try to analyze `coreutils` or `git` or `vim`, your (presumably) well-tested and working bounded model checker suddenly starts to misbehave in ways you never even thought possible. Not surprisingly, when we did this with Borealis, we encountered exactly that.

First, there were crashes. A lot of them. Many of them were caused by bugs in Borealis itself, as it had never previously encountered code configurations seen in these projects. Some are attributed to incomplete or incorrect information we receive from the Clang/LLVM pipeline. Yet other ones are caused by completely missing components never needed before, e.g., Borealis works as a drop-in replacement for a compiler, but many projects also need an archiver (`ar`) for a successful build.

We managed to fix or work-around most of these crashes, only to run into the next problem of real-world program analysis: performance. Different parts of Borealis, starting from SMT solvers to your everyday logging facilities, suffered immense slowdowns when working with **big code**, so we had to somehow optimize Borealis. Some optimizations are easy, as one can disable logging and boost the optimization level from `-O0` to `-O2`, getting a performance boost for free. Other optimizations are hard, making you profile your code again and again in search of fixable bottlenecks—the search only made harder by many external components with unpredictable performance.

As an example, when we started running experiments on `git`, it turned out 32 GBs of memory were not enough for the analysis to terminate. After aggressively optimizing for memory (and fixing a couple of logical memory leaks in the process), Borealis analyzed `git` successfully, after a week (sic!) of nonstop work. It was an achievement, but it was just not enough.

We started profiling the code, only to find out that parts of the system we never would have suspected were causing this slowdown. State slicing (used to prune the state of uninteresting predicates w.r.t. current safety property) was taking much too long compared even to the actual SMT procedure, because the alias analysis implementation we used was too slow. The predicate states exploded in size on `git` code, due to an inefficiency in our implementation which manifested only on deeply nested loops (unsurprisingly found in `git`). We even found some bugs in the functional programming library used extensively in Borealis.

It took us several months of trial and error, but we managed to fix these issues and reduce total `git` analysis time to a little over 5 h, i.e., 33× increase in performance. We still need to analyze the actual results and decide which paths we should explore next, but now it seems much more feasible than it did half a year ago.

## 7 Future Work

All in all, we do believe Borealis has successfully entered its adolescence but still has a long way to go. There are a lot of areas we would like to explore in the future; as of this chapter, our current work in progress includes the following:

- Using Borealis as a software repository mining tool to collect function code contracts via a combination of SMT and data mining methods
- Generating simplified function summaries based on source code heuristics
- Advancing our research on BMC-based test generation [29], e.g., improving support for complex data types
- Improving the quality of loop analysis and making the chains-of-recurrences approach work with recursion and loop-like function calls
- Designing a user-friendly function description language, which covers all aspects of [library] function behavior

Of course, we also have some long-term plans that are mainly targeted at making Borealis more of a software productivity tool than of a research prototype. When we have the resources, we would like to explore on how one might use BMC for implementing custom domain-specific analyses. We also need to continue testing

Borealis on difficult projects, so that it is stable enough to be used in continuous integration. As it is based on LLVM, we also think Borealis could be tweaked to work with other LLVM-based languages, such as C++, Rust, or Swift, and this potentially creates a plethora of interesting research questions.

## 8 Conclusion

In this chapter, we attempted to present all stages of research and development of our bounded model checking tool Borealis, from the basics to its evaluation on real-world programs. We discussed two of the most complicated problems in program analysis—loops and interprocedural interaction. We also presented our evaluation results on real-world programs and talked about the problems encountered and how we approached them. There are a lot of challenges ahead of us, still. Borealis needs to become more robust in terms of what code it can successfully analyze. We need to better our loop and interprocedural analyses. We also consider making Borealis into an extensible framework for domain-specific and user-defined analyses. Thankfully, all this does not seem impossible, now that we have dotted the *i*s and crossed the *t*s with this chapter.

**Acknowledgements** We would like to thank Bertrand Meyer for organizing the PAUSE conference and all who decided to spend their precious time reading this chapter.

## References

1. Akhin, M., Belyaev, M., Itsykson, V.: Improving static analysis by loop unrolling on an arbitrary iteration. Humanit. Sci. Univ. J. **8**, 154–168 (2014)
2. Akhin, M., Belyaev, M., Itsykson, V.: Software defect detection by combining bounded model checking and approximations of functions. Autom. Control. Comput. Sci. **48**(7), 389–397 (2014)
3. Akhin, M., Kolton, S., Itsykson, V.: Random model sampling: making Craig interpolation work when it should not. Autom. Control. Comput. Sci. **49**(7), 413–419 (2015)
4. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. Int. J. Softw. Tools Technol. Transfer **11**(1), 69–83 (2009)
5. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W.: Experiences using static analysis to find bugs. IEEE Softw. **25**, 22–29 (2008)
6. Bachmann, O., Wang, P.S., Zima, E.V.: Chains of recurrences — a method to expedite the evaluation of closed-form functions. In: ISSAC'94, pp. 242–249 (1994)
7. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, vol. 185, pp. 825–885. IOS Press, Amsterdam (2009)
8. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. Preliminary Design, version 1.4, 2008, preliminary edn. (2008)
9. Belyaev, M., Itsykson, V.: Fast and safe concrete code execution for reinforcing static analysis and verification. Model. Anal. Inform. Sist. **22**, 763–772 (2015)

10. Berlin, D., Edelsohn, D., Pop, S.: High-level loop optimizations for GCC. In: Proceedings of the 2004 GCC Developers Summit, pp. 37–54 (2004)
11. Beyer, D.: Status report on software verification. In: TACAS'14, pp. 373–388 (2014)
12. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS'99, pp. 193–207 (1999)
13. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI'06, pp. 427–442 (2006)
14. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT solver. In: CAV'08, pp. 299–303 (2008)
15. Buckland, M., Gey, F.: The relationship between recall and precision. J. Am. Soc. Inf. Sci. **45**(1), 12–19 (1994)
16. Calcagno, C., Distefano, D., Dubreil, J., et al.: Moving fast with software verification. NASA Formal Methods, pp. 3–11. Springer, Cham (2015)
17. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
18. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS'04, pp. 168–176 (2004)
19. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE'09, pp. 137–148 (2009)
20. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
21. Dan, A.M., Meshman, Y., Vechev, M., Yahav, E.: Predicate abstraction for relaxed memory models. Static Analysis, pp. 84–104. Springer, Berlin (2013)
22. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer, Berlin (2008)
23. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS'08, pp. 337–340 (2008)
24. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL'02, pp. 191–202 (2002)
25. Fratantonio, Y., Machiry, A., Bianchi, A., Kruegel, C., Vigna, G.: CLAPP: characterizing loops in android applications. In: DeMobile 2015, pp. 33–34 (2015)
26. Grosser, T.C.: Enabling polyhedral optimizations in LLVM. Doctoral dissertation (2011)
27. McMillan, K.L.: Applications of Craig interpolants in model checking. In: TACAS'05, pp. 1–12 (2005)
28. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: VSTTE'12, pp. 146–161 (2012)
29. Petrov, M., Gagarski, K., Belyaev, M., Itsykson, V.: Using a bounded model checker for test generation: how to kill two birds with one SMT solver. Autom. Control. Comput. Sci. **49**(7), 466–472 (2015)
30. Regehr, J.: A guide to undefined behavior in C and C++, part 1. http://blog.regehr.org/archives/213. Accessed 24 July 2017
31. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: HVC'11, pp. 160–175 (2012)
32. van Engelen, R.: Symbolic evaluation of chains of recurrences for loop optimization. Technical Report (2000)
33. Wang, X., Chen, H., Cheung, A., et al.: Undefined behavior: what happened to my code? In: APSYS'12, pp. 9:1–9:7 (2012)