C for Yourself: Comparison of Front-End Techniques for Formal Verification

Levente Bajczi
levente.bajczi@edu.bme.hu
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary

Zsófia Ádám
adamzsofi@edu.bme.hu
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary

Vince Molnár
molnar.vince@vik.bme.hu
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary

ABSTRACT

With the improvement of hardware and algorithms, the main challenge of software model checking has shifted from pure algorithmic performance toward supporting a wider set of input programs. Successful toolchains tackle the problem of parsing a wide range of inputs in an efficient way by reusing solutions from existing compiler technologies such as Eclipse CDT or LLVM. Our experience suggests that well-established techniques in compiler technology are not necessarily beneficial to model checkers and sometimes can even hurt their performance. In this paper, we review the tools mature enough to participate in the Software Verification Competition in terms of the employed analysis and frontend techniques. We find that successful tools do exhibit a bias toward certain combinations. We explore the theoretical reasons and suggest an adaptable approach for model checking frameworks. We validate our recommendations by implementing a new frontend for a model checking framework and show that it indeed benefits some of the algorithms.

CCS CONCEPTS

• **Software and its engineering → Software verification**; Formal software verification; • **Theory of computation → Parsing**.

KEYWORDS

Software Model Checking, Formal Verification, CEGAR, BMC, LLVM

ACM Reference Format:
Levente Bajczi, Zsófia Ádám, and Vince Molnár. 2022. C for Yourself: Comparison of Front-End Techniques for Formal Verification. In *International Conference on Formal Methods in Software Engineering (FormaliSE'22), May 18–22, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524482.3527646

FormaliSE'22, May 18–22, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9287-7/22/05…$15.00
https://doi.org/10.1145/3524482.3527646

1 INTRODUCTION

Formal verification of software with model checking has been evolving rapidly in the last decades, reaching a point where more and more often it can be applied on non-trivial real-world problems with acceptable performance. Many tools, however, suffer in another critical aspect: parsing and interpreting real-world programs. In the case of the C language, a wide range of language primitives, data types, and the sometimes loose specification cause many of the better tools to give up immediately during parsing the program, or produce incorrect results due to a misunderstanding of the specification.

More established software model checkers that are capable of tackling a reasonable part of the tasks in the Software Verification Competition (SV-COMP) [8] tend to solve the problem of parsing the input programs by building on existing compiler or parser frameworks (*frontends*). Most of these have been developed to compile C code into some intermediate representation or byte code tuned for efficient execution. Some of them (like LLVM – see Section 3.1) are really popular and mature, mostly because they provide a lot of optimizations and transformations that can simplify the program and its representation. For this reason, it is very tempting to exploit them in model checking as well, since one could expect that the optimizations and simplifications will help in making the analysis more efficient.

As the developers of a software model checking tool, we also followed this path for a while. During this journey, we have accumulated some experience on the pitfalls of this approach, mostly resulting from the fact that compiler technologies were obviously not optimized for producing outputs appealing to formal verification tools. We learned that in some cases, transformations like the one converting a program to Static Single Assignment (SSA) form [39] in LLVM may indeed help some algorithms, but will cause serious issues to others.

This has motivated us to review the data available from SV-COMP to see if developers of other tools have also faced this problem. We found that there is indeed a bias towards certain combinations – most prominently, tools using abstraction-based algorithms seem to avoid LLVM, while those based on bounded model checking (BMC) seem to benefit from the services of the more sophisticated compiler frameworks.

In this paper, we present our findings as three key contributions: *1)* we highlight the trend discovered among the SV-COMP competitors, rhyming to our own experience; *2)* we discuss the potential theoretical and practical reasons behind these observations based on characteristics of the employed algorithms and frontends and specify requirements towards an ideal solution; *3)* we validate our results with an experiment where we develop a new, custom frontend for our model checker tool and compare it to the original, rather well-established LLVM-based solution both with regard to abstraction-based and BMC-based algorithms. The result of the experiment confirms that even with such a makeshift solution, the performance of abstraction-based solutions can be increased significantly, although this approach voids the benefits of using compiler frameworks to parse a larger subset of C programs.

We hope that our results will foster discussion in the community about the best practices in the efficient development of software model checking tools for C programs, or even serve as a call for action to develop the analogue of tunable compiler frameworks for formal verification tools, potentially building on one of the already existing partial solutions.

The paper is structured as follows. In Section 2, we present the necessary background on software model checking. In Section 3, we examine the software verification tools and their choice of frontend projects. In Section 4, we present a proposal for the design of a verification-centric frontend framework, for which we also evaluate a proof of concept implementation in Section 5. Finally, we summarize our findings in Section 6.

2 SOFTWARE MODEL CHECKING

Model checking [21] is a mathematical method of verifying that a formal model adheres to a formal specification. To achieve this, the model checking algorithm will analyse the *state space*, i.e., the set of all possible configurations exhibited by the model. An exhaustive enumeration of the state space is often not feasible due to the phenomenon called *state space explosion*, which refers to the number of states growing larger than practically manageable (or even infinite). Therefore, model checking algorithms generally use smart exploration strategies and heuristics, or a special encoding, often coupled with abstraction.

In software model checking, the input is a program code instead of a formal model. Even in this case, model checking needs a formal representation of the specified behavior, as programming languages tend to have ambiguities and complex structures that may help the developers express themselves more concisely, but are hard to handle in a low-level algorithm. To bridge the gap between formal models and program code, developers of software model checking toolchains use separate *frontends* before running the analysis (often called *backend* in contrast), which transforms the input program into a suitable formal model. In this paper, we use *Control Flow Automata* (CFA) as the intermediate formalism, a representation widely used in software model checkers [12].

Definition 2.1 (Control Flow Automaton). A CFA is a 4-tuple (V, L, E, l_0) [12], where

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of variables with domains $D_{v_1}, D_{v_2}, \ldots, D_{v_n}$,
- $L = \{l_0, l_1, l_2, ...\}$ is a set of locations, modeling the program counter,
- l_0 is the initial program location, the entry-point of the program,
- $E \subseteq L \times Ops \times L$ is a set of edges, which represent the executed operations between two locations. *Ops* can be:
  - assignments (e.g., $x := y + 2$),
  - assumptions (e.g., $[x = 0]$), or
  - nondeterministic assignments, *havocs* (e.g., *havoc* x)

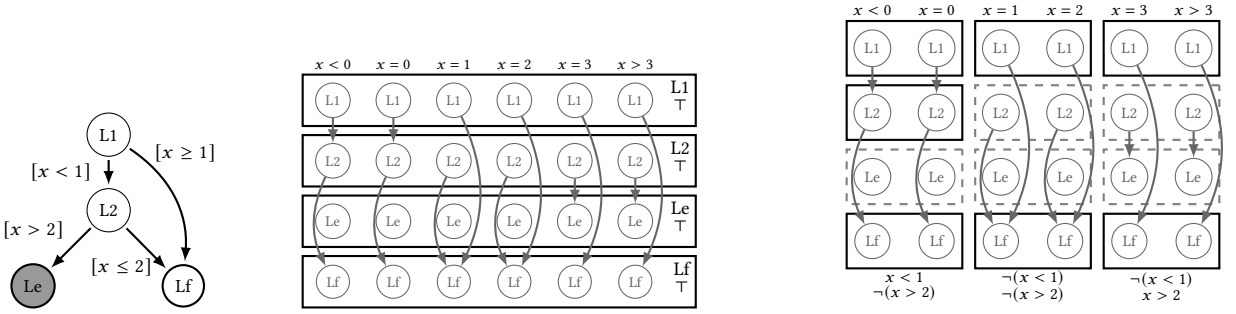
It is important to distinguish between *locations* and *states* of a CFA. A *state* in the CFA consists of a *location* and a system of constraints over the variables present in its operations. If such constraints are valuations that assign a value to each variable, then the state is a *concrete state*. Otherwise, the state is an *abstract state*, corresponding to a number of concrete states for which the constraints hold.

An execution of the input program corresponds to a *concrete trace*, which is a series of concrete states and transitions in the state space of the CFA. Observe that these traces can be projected onto the graph induced by the locations and edges of the CFA to get a directed path, which correspond to the execution of the related parts of the program. Similarly, an *abstract trace* is a series of abstract states and transitions in some abstraction of the state space, which also maps to the graph induced by the CFA. However, abstract traces do not always represent executions of the program. We say an abstract trace is *feasible* if applying the effects of the transitions on each abstract state do not contradict the target abstract state. Otherwise, the trace is *infeasible*. Feasibility depends on the existence of a concrete trace which follows the same path as the abstract trace, and each concrete state satisfies the constraints of the corresponding abstract state.

Checking the feasibility of a trace can be formulated as an SMT (Satisfiability Modulo Theories) query, which in turn can be evaluated by an SMT solver. As SMT formulae use *constants* rather than *variables*, the trace needs to be transformed into a Static Single Assignment (SSA) form [39]. An SSA form allows at most one assignment to each variable before its first use by further expressions, and none thereafter. This makes variables effectively constants, making them suitable for SMT expressions. To achieve this transformation, we change the variables along the trace to *indexed constants*, based on the current *indexing* of the variables. An indexing is a function that maps indices (integers) to variables. Initially, all indices are 0, and they gradually increase every time the corresponding variable is assigned a new value. Therefore, assumptions can be automatically used as SMT constraints with the current indexing, while assignments can be transformed into an expression formulating the equality of the next indexed constant of the left-hand side variable with the expression on the right-hand side. *Havocs* only increase the index of the variable, and do not constrain its value, as expected from a nondeterministic assignment.

2.1 Reachability in the CFA

The goal of software model checking is to either produce a *counterexample* contradicting the specification, or to prove that the specification will always hold. In this paper, we assume that the specification solely contains *reachability* queries, which designate



(a) CFA **(b) State space over explicit-value abstraction** **(c) State space over predicate abstraction.**

Figure 1: Concrete states are circles, rectangles denote abstract states, dashed rectangles are unreachable.

unsafe states in the state space of the input program. A counterexample shows a feasible trace from an initial state to an unsafe state, and a proof of safety needs to show that no unsafe state is ever reachable. While in theory reachability queries can designate any state *unsafe*, it is customary to further restrict this to *unsafe locations* due to the specification often being given as *assertions* in the code. Therefore, the CFA is extended with a predicate which partitions the locations to a safe and an unsafe subset. If an execution reaches a state that belong to an unsafe location (i.e., an unsafe state), then the input program is *unsafe*; if there is no execution that can reach an unsafe state then the input program is *safe*.

2.2 Approaches to Scalable Model Checking

As discussed earlier, enumerating all states and checking if an unsafe state is reachable is not feasible due to the large number of states. In this paper, we look at two widely used solutions to the state space explosion problem: Bounded Model Checking (BMC) [15] and Counterexample-Guided Abstraction Refinement (CEGAR) [20] as they are prevalent in software model checking.

2.2.1 BMC. The idea behind BMC is that the most common bugs are reachable within the first k steps of an executing program [15]. Therefore, the state space of the input model is explored only up to a bound k, and if an unsafe state is explored, the input model can be considered erroneous without having to explore all possible states. The disadvantage of the approach is that a proof of safety is produced only if the diameter of the state space is at most k.

There are extensions to this approach such as the application of k-induction [10, 25], which in general still constrain the search space but often yield better results. These techniques usually rely on SMT solvers, encoding traces as SMT formulae in conjunction with the formula describing the reachability query. The satisfaction of such formula signifies a counterexample, while unsatisfiability means there are no reachable error states within the given bound. When applied to CFAs, the formula often consists of several descriptions of traces along different paths in the CFA.

Other approaches in software model checking that are sometimes labeled as BMC operate with loop unrolling, where loops of the program are unrolled up to k iterations and executions iterating more than k are not considered. The common feature of BMC approaches is that they deal with finite traces and therefore do not have to cope with potentially infinite iterations.

2.2.2 CEGAR. Abstraction-based methods aim to also provide a way to prove safety without enumerating the entire state space. Instead of covering a certain number of states explicitly, abstraction is used to handle groups of states together, which may greatly reduce the number of abstract states in the (abstract) state space [20]. The rules that govern which concrete states shall be grouped together are called the *precision* of the abstraction. Transitions among the abstract states are constructed in the following way: if any two concrete states are connected via a transition in the concrete state space, then the abstract states containing those states must be connected via the same transition as well. This provides a *safe overapproximation*, as every possible behavior observable in the concrete state space is also observable in the abstract state space. If an abstraction is found without a feasible trace to an unsafe state, the original model must be safe as well [20]. However, a trace in the abstract state space is not necessarily feasible over concrete states. Therefore, an abstraction-based technique needs to have a strategy to *refine* the abstraction until a precision is found with which the abstract state space either contains a feasible trace (a *counterexample*) or does not contain any abstract trace leading to an error state.

To solve the problem of finding bugs, the abstraction needs to be *refined* until such a precision is found that either contains no path to an unsafe state (when the input model is *safe*) or a feasible trace is found – in which case the input model is *unsafe*, and the concrete trace serving as the proof of feasibility also serves as a counterexample. Should a trace be found that is infeasible, a new precision is necessary that distinguishes the concrete states which caused the infeasible trace to appear. The method of continuously constructing the abstract state space and then refining the precision based on the infeasible counterexample is called *Counterexample-Guided Abstraction Refinement* (CEGAR) [20].

Counterexample-Guided Abstraction Refinement (CEGAR) [20] uses infeasible (or *spurious*) counterexamples to refine the abstraction iteratively. The goal of the refinement is to modify the precision such that the spurious counterexample does not appear in the next

	CPAChecker based tools (3x)	Ultimate Family (3x)	Gazer-Theta	Symbiotic	ESBMC (2x)	Frama-C	Divine	Dartagnan	Smack	2LS	CBMC	Pinaka	Goblint	Korn	Lazy-CSeq	PredatorHP
CEGAR	X	X	X													
BMC	X		X		X			X	X	X	X	X			X	
Eclipse CDT	X	X														
LLVM (optionally with Clang)			X	X	X	X	X									
SMACK (LLVM)								X	X							
Cprover Framework (Goto-CC)										X	X	X				
Other													X	X	X	X

Table 1: The tools of SV-COMP 2021 and the analyses as well as frontend technologies they employ.

abstraction, which means that the new precision has to split an abstract state. CEGAR uses the concept of *abstract domains* to govern the types of rules the precision may have [30].

Definition 2.2 (Abstract Domain). An *abstract domain* is a tuple $D = (S, \top, \bot, \sqsubseteq, expr)$ [30], where:

- S: Lattice of abstract states (possibly infinite)
- $\top \in S$: Top element
- $\bot \in S$: Bottom element
- $\sqsubseteq \subseteq S \times S$: Partial order over the lattice S
- $expr$: A mapping from an abstract state to a constraint on the state variables (i.e., an expression)

For the purposes of this paper, we present two generally well-performing domains that will be used in our experiment: explicit-value abstraction and Cartesian predicate abstraction.

Explicit-Value Abstraction. The *explicit* abstraction domain defines the precision as a set of *tracked* variables. Formally:

- $\forall s \in S$: A variable assignment of each *tracked* variable to a value of its domain, extended with top (arbitrary value) and bottom (no assignment possible) elements.
- $\top \in S$: No specific value is assigned to any of the tracked variables.
- $\bot \in S$: No assignment is possible to the tracked variables.
- $\sqsubseteq \subseteq S \times S$: $s_1 \sqsubseteq s_2 \iff (s_1 = s_2) \vee (s_1 = \bot) \vee (s_2 = \top)$.
- $expr$: A conjunction of constraints binding the values of tracked variables for each abstract state.

Predicate Abstraction. The *Cartesian predicate* abstraction domain defines the precision as a set of tracked predicates. Formally:

- $\forall s \in S$: A conjunction of predicates or their negations.
- $\top \in S$: The predicate *True*.
- $\bot \in S$: The predicate *False*.
- $\sqsubseteq \subseteq S \times S$: $s_1 \sqsubseteq s_2 \iff (s_1 \implies s_2)$.
- $expr$: A conjunction of the predicates or their negations constituting the given state.

For practical reasons, locations are generally tracked explicitly no matter what abstract domain is used. This helps the algorithms track the possible transitions from an abstract state, as only the outgoing edges of one location need to be explored.

For an example, consider Figure 1. The CFA in Figure 1a shows a *safe* program, because intuitively no data state may exist that both satisfy $x < 1$ and $x > 2$. However, an explicit analysis (with a limited amount of memory) will never be able to enumerate all values of x, because theoretically an infinite amount could exist – and even practically, a single integer will have billions of possible values. Therefore, as seen in Figure 1b, the explicit abstraction denotes the data state as $\top$, meaning no information on x is recorded. As both $L1 \rightarrow L2$ and $L2 \rightarrow Le$ exist among the concrete states, the abstract error state will be reachable, albeit via an infeasible path. As refinement is no longer possible (all variables are currently tracked), the analysis cannot proceed to prove correctness.

In comparison, if a predicate abstraction-based analysis has found the predicates $x < 1$ and $x > 2$, it can group the concrete states into three partitions, as seen in Figure 1c. As none of the partitions consider *Le* reachable, the analysis can conclude the *safety* of the input program.

3 ANALYSIS OF THE STATE OF THE ART

As our first contribution, we analyse the data available about the tools that competed in SV-COMP 2021 [8] to identify the most commonly used frontend technologies for parsing C programs, pairing this data with the software model checking strategies employed by each tool (focusing on those discussed in Section 2).

On SV-COMP 2021 there were 30 participating tools, 6 of which are Java verifiers, 2 are portfolio verifiers using other tools, and sufficient information was not available about the tool Brick regarding the employed frontend technology. Table 1 summarizes our findings about the remaining 21 tools. Note that some tools are not shown explicitly: CPA-BAM-BnB [2], CPALockator [3], CPAChecker [13] are grouped as CPAChecker-based tools and UAutomizer [33], UKojak [26] and UTaipan [29] are grouped under the Ultimate Family column. ESBMC also has two separate variants, ESBMC-kind [27] and ESBMC-incr [22], the latter of which is developed specifically for the verification of multithreaded programs.

Section 3.1 gives an overview of the most prevalent frontend technologies, while Section 3.2 provides our conclusions based on the data.

3.1 Frontend Technologies

3.1.1 Eclipse CDT Parser. The Eclipse C/C++ Development Tooling (Eclipse CDT) project[1] provides a fully functional IDE for C/C++ development. Among the various features provided by this IDE are the C and C++ parsers that are of primary interest when used as a

[1] https://www.eclipse.org/cdt/

frontend for verification[2]. These parsers are available as standalone components and can be used to generate Abstract Syntax Trees (AST) [11], although using this feature in practice carries some technical difficulties.

The tools using the Eclipse CDT parser are the CPAChecker-based tools and the Ultimate tool family.

3.1.2 LLVM. The LLVM project[3] [36] is a well-known collection of different modular compiler technologies, supporting arbitrary programming languages, but best known for C/C++ related tools and modules. One of the most well-known features is the LLVM Intermediate Representation (LLVM IR). LLVM IR is a human-readable, typed assembly language and in-memory representation aiming to provide type-safety and flexibility. It uses a Static Single Assignment (SSA) representation.

The LLVM project provides a large set of *passes*, i.e., optimization and transformation steps to be executed on LLVM IR. These optimizations were created from a compiler's standpoint, i.e. to make the program run as efficiently as possible. Clang is the "LLVM-native" C/C++ Compiler tool of the project, which uses the LLVM Intermediate Representation (LLVM IR) with the various passes and is capable of outputting the IR itself instead of an executable.

Tools using LLVM (with or without clang) are Divine [6], Symbiotic [18], ESBMC-incr [22] and ESBMC-kind [27], Frama-C [23], and Gazer-Theta [1].

3.1.3 SMACK. SMACK [38] is both a modular software verification toolchain and a self-contained software verifier. It is capable of translating LLVM IR to Boogie [7], a popular intermediate verification language. It is used by several tools as a frontend as Boogie was specifically designed to simplify the implementation of verification algorithms. However, the SSA representation originating in LLVM is transferred into the Boogie code.

Tools building on the utilities of SMACK are SMACK [38] itself and Dartagnan [28].

3.1.4 Goto-CC. Goto-CC[4] compiles C/C++ code to "GOTO" programs (i.e. control-flow graphs) to be used mainly by verification or testing tools (i.e. GOTO programs are not meant to be executed). It is part of the CProver framework, a software verification platform known mainly for CBMC, a BMC verifier for C and C++ programs. It can handle the build system of large projects replacing either *gcc* or *Microsoft's Visual Studio compiler*, sparing the verification tool from implementing this often cumbersome task.

Tools using Goto-CC are CBMC [35], 2LS [16] and Pinaka [19].

3.2 Discussion

Out of the 21 tools only 4 uses a frontend other than the ones introduced in Section 3.1, and most of these 4 are not BMC or CEGAR-based tools. Based on Table 1 there is a clear bias towards the Eclipse CDT parser in CEGAR-based tools, whereas the remaining verifiers are distributed between LLVM and Goto-CC, with 7 out of 10 building on an LLVM-based solution.

Next, we provide our interpretation and examine the potential theoretical reasons behind the patterns shown by the data. Our starting hypothesis is that tools have a motivation to use more advanced and well-established compiler frameworks with a lot of optimizations and utilities and a wide support for language constructs. Therefore, we assume the deviation is when tools choose a less advanced framework such as the Eclipse CDT instead of LLVM.

Based on our experience as developers, we place the focus on the effect of the SSA form inherent in the LLVM IR. The following sections summarize the consequences of this transformation with regard to BMC and CEGAR.

3.2.1 Models in SSA for BMC. As introduced in Section 2.2.1, BMC encodes the explored paths in an SMT formula, which uses constants and thus benefits from an SSA form. By using LLVM as a frontend, the transformation step to such an SSA format is already implemented and applied while the model transformation happens.

Furthermore the long years of work and many active users of the optimizations of the LLVM project ensure a level of quality that cannot be easily replicated and these optimizations are executed on the SSA format (i.e. on the LLVM IR) already.

Another advantage of the LLVM project is that it is comprehensive in the sense that many non-trivial and complex semantic rules of the C language are automatically transformed into the much more straightforward format of LLVM IR, e.g., integer promotion, overflows with wraparounds, implicit casts and operator precedence are already handled by clang, making the implementation of model transformation easier.

These capabilities make the LLVM project the most common choice for BMC tools, especially when an out-of-the-box solution is beneficial.

In contrast to LLVM, Goto-CC was made specifically with verification in mind. Although it does execute some necessary transformations to make the input unambiguous, it does not utilize aggressive optimizations, rather it just creates a fairly verbose CFA out of the code. Tools capable of parsing this CFA can verify it directly or implement and apply some further transformations, e.g. optimizations or simplifications on it.

This makes Goto-CC another common choice, as it provides more customizability in exchange for losing the optimization passes of LLVM.

3.2.2 Models in SSA for CEGAR. One of the key features of CEGAR is utilizing an abstract domain over the variables of the input task to tackle state space explosion. Ideally CEGAR converges to such an abstraction level where the feasibility of a proof or a counterexample is verifiable in a reasonable amount of time, e.g., finding predicates that are enough to show that the input task is safe without enumerating all possible values.

In software model checking, there is often a semantic meaning or pattern hidden behind variables [24] and the effect of these patterns when using different abstract domains can even be used to aid verification [4]. If variable patterns get lost during the initial model transformation, CEGAR might have a much harder time finding an optimal abstraction level. Therefore, any frontend that erases this information will be disadvantageous for the verification workflow.

One such frontend is LLVM, where the SSA format burdens variable patterns from two separate perspectives: first, multiple SSA variables express the original role of a single variable (and

[2]https://wiki.eclipse.org/CDT/designs/Overview_of_Parsing
[3]https://llvm.org/
[4]https://www.cprover.org/Goto-CC/

hence a refinement using a single SSA variable might be too fine-grained); and second, further optimization passes might make a single SSA variable refer to multiple variables in the source file. This also eliminates the solution of mapping the variables to their SSA counterparts and only allowing refinement via groups of variables – which will also introduce too many variables in one iteration, making the resulting level of abstraction too low and therefore suboptimal.

Furthermore, the SSA-representation of a program tends to have more variables than its source, even when aggressive optimizations are used. This number itself can hinder the performance of CEGAR, as more iterations are necessary to achieve the same level of abstraction, and the precision will contain too many variables. For example, as we have seen in Section 2.2.2, predicate analysis is generally more expressive than using explicit-value abstraction – but predicate abstraction relies on finding a small set of expressions to track, as too many will result in poor performance.

The same problem will not burden BMC, as it already utilizes an SSA format and does not rely on abstraction. The trivial solution to this issue is finding an approach, which transforms the input program directly from the code, eliminating the SSA step.

What we have seen in Table 1 is that CEGAR-based tools are not that likely to use LLVM as their frontend as other tools. Both large CEGAR verifier families employ the Eclipse CDT project instead.

The C and C++ parser of the Eclipse CDT project was designed mainly for performance. It handles preprocessing and then creates an AST for each translation unit, which can be processed through its API. It handles ambiguity, but does not compute type information. It is an "all-in-one" solution handling the difficult tasks of parsing C out of the box. The disadvantage of this is that the verification tool might want to handle these difficulties and corner cases by itself instead of the pre-defined way, warranting a parser which is less monolithic and works on a lower abstraction level.

4 DESIGNING CEGAR FRONTENDS

Based on our findings, we established that an end-to-end frontend project like LLVM would be beneficial to the software model checking community, without its SSA-format. We propose the following requirements for a frontend framework built foremost for verification purposes:

- **R1 [Formal]** The resulting model shall be mathematically precise and algorithmically easy to interpret
- **R2 [Configurable]** The transformation process shall be configurable in terms of handling undefined elements of the source language
- **R3 [Direct Access]** The analyses must have direct access to variables in the C code
- **R4 [Verifier-Centric]** The resulting model should be optimized for verification, not for executable generation
- **R5 [Metadata Access]** The verification tools must have access to metadata from the source file
- **R6 [Unhandled Patterns]** The transformation must raise an exception if an unhandled pattern is found, lest an erroneous model be created

R1 [Formal] is generally applicable to all formal models as mentioned in Section 2, and all examined frontends fulfil this criterion.

R2 [Configurable] is necessary to deal with ambiguities in the source language transparently, as hiding these details might cause unwanted results. At the minimum, a warning has to be issued to the user that the input program contains under-defined patterns.

The rationale for **R3 [Direct Access]** is well established by Section 3.2.2: if a CEGAR-based analysis does not have direct access to variables, discovering the underlying patterns either takes more iterations and therefore time, or this information is simply lost.

R4 [Verifier-Centric] is based on the observation that conventional compilation and software verification rely on different optimization techniques. For example, bit-precise expressions are easy to execute on an actual processor, while reasoning about them is very resource-intensive for SMT-solvers [17].

R5 [Metadata Access] is more applicable to the verification workflow than the analysis itself, as producing a counterexample is arguably more important for the developers than determining safety. This way, the trace causing the unsafe behavior to appear is clearly in front of the user, which can be used to track down and eliminate the bug. To achieve this, the verification tool needs to map the output of the analysis (i.e., a trace in the formal model) to syntactic elements of the source file.

R6 [Unhandled Patterns] addresses the issue of completeness. While it might be possible to handle all elements of the input programming language, it is certainly advisable to define a smaller subset which is supported in its entirety, rather than try and handle every corner-case by itself. Such constraints are common in the domain of safety critical systems, as more conventional quality assurance methods (e.g., certification) require a clearly defined input language as well [32]. If an input does not follow the constraints of this subset, the parser needs to raise and exception and terminate parsing, as the resulting model is almost certainly erroneous.

As shown before, different analyses rely on different preprocessing steps. Therefore, we divide the workflow into two major stages. First, the input program is mapped to a so-called *verbose* CFA, which is common to all analyses. Then, the verbose CFA is further transformed and optimized, resulting in the final CFA, which will serve as the input model to the analysis backend. This way, a potential user can easily swap out parts of the workflow to better suite their needs. Note that the process is described for a single source file, where pre-processing has already taken place.

4.1 Input to Verbose CFA

To fulfill **R3 [Direct Access]** and **R5 [Metadata Access]**, the transformation process needs to work on the syntactic level of the source code. This way, the process can be sure to include all necessary metadata (e.g., line numbers and variable names) in the model. Furthermore, we have to define a mapping from each element of the input language to a model element of the resulting verbose CFA.

In order to handle all elements properly, the formalism of a verbose CFA has to be extended with an equivalent of C-style functions. This can be done by defining the verbose CFA as a *collection* of simpler CFAs (i.e., *procedures*), each having a list of parameter declarations along with directions *in* or *out* for input and output parameters (i.e., return value(s)). Furthermore, the edges of the simple CFAs have to allow procedure invocations as well, which

specify expressions for the input parameter bindings, and variables for the return values.

Even though we could follow the same pattern and apply it to other elements of the C language (e.g., a separate statement for a do-while loop), we have to pay attention to **R1 [Formal]**: if the resulting model is not a simple enough representation of the input, it is superfluous to apply the transformation in the first place.

R2 [Configurable] is only relevant at this stage, because the C standard contains many points of uncertainty, while the resulting verbose CFA should be unambiguous. As an example, the data type `char` is not defined to be `unsigned` or `signed` when declared without a signedness qualifier. This not only determines the meaning of a bit-pattern in a `char` variable, but also specifies if a wrap-around should occur when the value is too big to fit in the designated memory location – in the case of an `unsigned char`, $255 + 1$ will result in 0, but for a `signed char` the result of $-128 - 1$ is undefined [34], which is another ambiguity as well. Our advice is to follow the patterns of well-established projects such as GCC[5] or Clang/LLVM [36] by default, but *warn* the user about this occurrence and allow for an easy redefinition of its handling – in this case `char` is `signed`, and overflowing `signed` values wrap around as their twos-complement representations.

Perhaps the most important underspecified part in C is the width of integer types. Minimum widths and a strict sequence of size (*rank*) is given for integers, but their precise values are left to be implementation-specific. To deal with this uncertainty, the transformation has to be configurable to support any legal architecture. For example, the competition rules of SV-COMP [8] specify the use of two *data size neutral* programming models defined by the Single UNIX Specification[6], *ILP32* and *LP64*. These are safe default values, but custom architectures need to be supported as well. LLVM (through Clang) only allows named programming models by default, and has to be ported to any other architecture – which is a disadvantage against source-level frontends such as Eclipse CDT or goto-cc, which allow later steps in the workflow to use a custom programming model.

4.2 Optimizing the CFA

The verbose CFA is already a formal model, but its verbosity makes it suboptimal for the analyses. Depending on the capabilities of the utilized analysis, function calls or pointers (among others) might not be supported natively, and therefore a pre-processing step is necessary to clean up the CFA. Furthermore, size reduction and other graph transformations might also be necessary to make the model verifiable in a timely manner – due to the iterative nature of CEGAR, if an optimization can be done prior to starting the analysis, the performance improvement might show up in each iteration and therefore multiply its effectiveness.

Building on the architecture of the LLVM project [36], the optimization stage is further divided into *passes*. Each pass is a separate (and mostly independent) graph transformation step, which are executed sequentially over the verbose CFA. The resulting model is no longer a *verbose* CFA.

[5]https://gcc.gnu.org/

[6]https://unix.org/whitepapers/64bit.html

Based on their desired effect, passes can be grouped into two categories: *elimination* passes transform unwanted model elements into equivalent representations over supported elements, and *optimization* passes change model elements deemed too verbose into more concise forms. Most of these passes will be tool- and analysis-dependent, but we introduce some that generally applicable to the workflow.

4.2.1 Function Inlining. If an analysis only supports single-procedure CFAs (as the conventional definition does not allow procedure invocations), the procedure calls need to be inlined, i.e., substituted for the definition of the procedure. As an example for an *elimination* pass, this can be done iteratively: starting from the main procedure and only inlining the functions to a single layer, repeating the process until no more procedure invocation is found. However, in the case of recursive programs, this method will not terminate, and therefore an exception needs to be raised for the user (as per **R6 [Unhandled Patterns]**).

A note on **R6 [Unhandled Patterns]** and external (or otherwise undefined) function calls in C. The verbose CFA already allows procedure calls on edges – so allowing non-specified C-functions to remain as non-specified procedure invocations seems trivial. *Out* parameters can be *havoc*-ed (i.e., nondeterministically assigned), and *in* parameters discarded. However, this might produce false positive results: consider the `fabs(f: float): float` standard library function. Without prior knowledge, the statement $-1.0f == fabs(-1.0f)$ is SAT, because a nondeterministic `fabs()` might as well return the value it was passed. However, in practice, the value is never a negative number, and therefore a seemingly feasible trace might be actually infeasible – the reason why **R6 [Unhandled Patterns]** is important.

The solution to this problem relies on building a standard library of functions for the frontend as well. Besides covering the functions in the C standard library, precedence exists for such collections of functions in more specific circumstances as well. For example, SV-COMP specifies several custom functions to deal with nondeterminism, or signal the start of an atomic block [8]. If the transformation process strictly follows **R6 [Unhandled Patterns]**, encountering any unknown function has to raise an exception – which also warrants the easy specification of an allow-list of functions, should the user decide that the function may be modelled nondeterministically.

4.2.2 Nondeterministic Values. As mentioned above, it might be beneficial to model nondeterministic values in the model (such as user input). However, C and the compiler imposes implicit constraints on such values by defining their signedness and size – details that need to be preserved when the expressions are encoded for the SMT-solvers. For example, the trace $havoc\ x; [x == -1]$ might be feasible if x was a `signed int`, but infeasible if it was an `unsigned char`. One possible solution is to use bit-precise reasoning, which hinders performance [17]; or assumptions asserting the ranges of the data type have to follow each *havoc* statement to constrain its value.

4.2.3 Large-Block Encoding. As an example for a pure *optimization* pass, creating a large-block encoding variant of the CFA can greatly reduce the verification time of a task [9]. Large-block encoding

replaces sequential edges with a single edge containing an ordered conjunction of their statements, and parallel edges with an edge containing a disjunction of their statements.

4.2.4 Removing Dead-Ends. Some branches of the CFA may not contain any directed path toward an unsafe state after the initial transformation. These sections can safely be discarded, as their exploration will never result in a change of verdict – if other parts of the program contain bugs then the program is erroneous, and if all other parts are safe then the entire program is also safe.

5 EXPERIMENTAL EVALUATION

To support our previous claims, we devised an experiment that examines the behavior of an LLVM- and a grammar-based frontend when combined with BMC and CEGAR. To achieve a fair comparison, we implemented the two frontends in the same verification framework [40], which has access to both BMC- and CEGAR-based analysis backends. We do not intend to compare the BMC and CEGAR algorithms with each other, but rather the frontend techniques used and their effect on the performance of the algorithms.

As CEGAR is a highly configurable family of verification algorithms, its exhaustive comparison to BMC is not feasible. To demonstrate the practical applicability of the presented approach, we show that two fundamentally different applications of CEGAR (namely, one based on explicit-value abstraction and one based on predicate abstraction) both exhibit the same biases towards frontend choice.

We intend to answer the following research questions:

RQ1 Does BMC benefit from a frontend with a transformation to an SSA format compared to a more direct, grammar-based approach?

RQ2 Does CEGAR with explicit-value abstraction benefit from the grammar-based frontend compared to the frontend with a transformation to an SSA format?

RQ3 Does CEGAR with Cartesian predicate abstraction benefit from the grammar-based frontend compared to the frontend with a transformation to an SSA format?

5.1 Implementation

We chose to implement a proof-of-concept version of the frontends in our verification framework, which is built mainly around abstraction-refinement, i.e., CEGAR [20] techniques. For this work we also added a BMC algorithm to the framework's repertoire using an iteratively deepening BMC algorithm [10]. The framework does not have a native C-frontend, but it has been extensively validated using other sources of formal models and external frontends for C verification. Native support for CFAs was already present, but an extended version had to be added to conform to the requirements of the *verbose* CFA.

5.1.1 Frontend Implementation. The LLVM- and grammar-based frontend implementations build on LLVM 11.0[7] and ANTLR [37], respectively. *Grammar-based C frontend* is an own C frontend of the tool, using an ANTLR grammar to parse and transform the program into a verbose CFA. Furthermore, custom passes are available to optimize the resulting model, utilizing the approach introduced in Section 4. The choice fell on ANTLR instead of a more integrated solution such as Eclipse CDT or Goto-CC due to the lower-level access to the source file, eliminating all black boxes that could hinder verification or counterexample generation. *LLVM-based C Frontend module* is a small external library, serving as a lightweight LLVM-based parser (while also applying optimizations on the LLVM IR). Its goal was to leave only the CFA generation steps in the verification framework and move everything else (parsing, optimizations) into the native module to better utilize the capabilities of LLVM.

Both frontends use passes to transform an intermediate representation: the LLVM-based one uses both built-in and custom LLVM passes, and the ANTLR-based uses only custom ones. Both use a version of the following:

- Function inlining
- Nondeterministic input transformation
- Dead code removal
- Various expression simplification passes

Besides these, LLVM has access to more advanced simplification and transformation techniques such as `simplifycfg` for simplifying the control flow and `sroa` (scalar replacement of aggregates) to break up complex structures such as arrays and structures into individual primitives, greatly reducing the workload of the SMT solver. On the other hand, the ANTLR-based frontend has access to a large-block encoding algorithm that LLVM lacks [9]. The summary of the two workflows can be seen in Figure 2.

5.1.2 Backend Configurations. As *CEGAR* is a highly configurable algorithm and success rates can deviate heavily between configurations, we used two generally well-performing configurations from the predicate and the explicit domain, using backwards binary interpolation and sequential interpolation, respectively. For a more detailed description, refer to [31].

Our implementation of *Bounded Model Checking* utilizes an iteratively deepening bound to work around the issue of choosing a bound, which is either too small or too large. Thus the practical limit stopping the analysis is the time limit, which is the same for each configuration (900 seconds).

5.1.3 Input Programs. As input we use a subset of the benchmarking set of SV-COMP[8], namely a number of sub-categories from the C tasks of the ReachSafety category.

As the frontends have differences in what language elements they support, we used only those SV-COMP subcategories where both should be capable. We excluded the categories Arrays, Floats, ProductLines (containing combined structs and function pointers), Loops, Heap, Recursive and Combinations (also containing a large number of floating point operations).

Among the 2115 chosen input tasks there is a similar amount of safe and unsafe tasks (namely 1117 safe and 998 unsafe programs).

5.2 Execution Environment

The benchmarks were executed with the benchmarking framework of SV-COMP, *Benchexec* [14] version 3.8, providing reliable measurements on resources. The *ANTLR-based frontend* is built into

[7] https://releases.llvm.org/11.0.0/

[8] The commit we used: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/99d37c5b4072891803b9e5c154127c912477f705

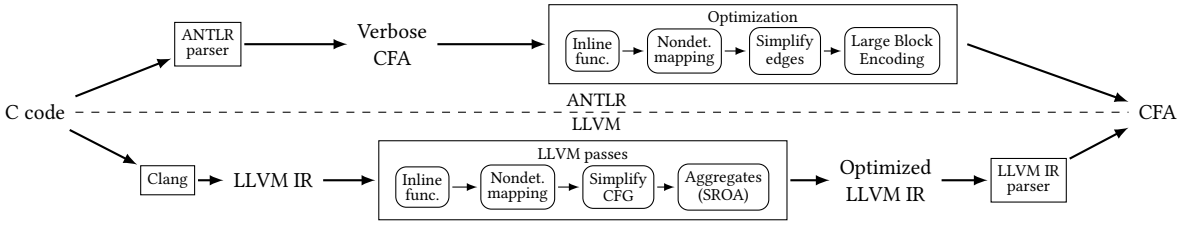


Figure 2: Summary of the workflow for the grammar- and LLVM-based frontend implementations

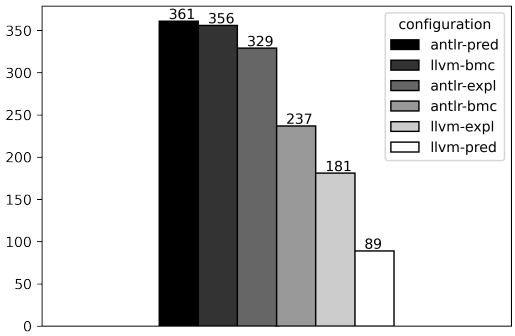


Figure 3: Number of successful results per configuration

the verification framework, while the *LLVM-based frontend* is a separate module. The experiment was executed on virtual machines running *Ubuntu 20.04.2 LTS* with Java *OpenJDK 11.0.13*.

There was a memory limit of 15GB and a CPU core limit of 8 cores set. There was also a CPU time limit of 900 seconds per task, the same limit as the CPU time limit used on SV-COMP.

5.3 Evaluation

In Figure 3 the number of successful analyses is shown for each algorithm-frontend pair. Each configuration was executed on the same 2115 tasks, as described in Section 5.1.3, with a 15 minute CPU time limit per task.

5.3.1 Evaluation on BMC. RQ1 asked if BMC benefits from using the frontend creating a model in an SSA representation compared to a more direct, grammar-based approach.

The BMC algorithm using the LLVM based frontend did not just outperform the BMC-ANTLR pair, but it even came up as second best, showing the power of both the LLVM IR and the optimizations. BMC with LLVM outperformed BMC with ANTLR by 119 tasks, resulting in a 40.13% difference.

5.3.2 Evaluation on CEGAR. The questions RQ2 and RQ3 asked if CEGAR with Cartesian predicate and explicit analysis benefits from using the ANTLR-based frontend with a more direct transformation compared to a frontend transforming the input into an SSA format. Both CEGAR analyses using the ANTLR frontend performed really well. Predicate analysis finished in first place and explicit value analysis in third place globally.

The same algorithms became the worst performing configurations when executed with the LLVM frontend. The predicate analysis solved the least tasks from all configurations – indicating that it might be more sensitive to losing information due to SSA.

What makes these results even more convincing is that the LLVM project and its optimizations are constantly being polished by a broad user-base, while the optimization steps implemented in the grammar-based frontend are much less mature.

5.3.3 Conclusion of the Experiment. The results of the executed benchmark line up with our experiences and conclusions detailed in this work: there is a loss of performance if using CEGAR with a frontend which transforms the input program to an SSA format.

There is also a difference in this loss depending on the abstract domain: while both explicit and Cartesian predicate analysis seem to be affected, predicate analysis suffers a more significant loss when receiving models in an SSA format. This is as expected, because both the larger number of variables and their single use nature make it more difficult (or even impossible) to deduce suitable predicates.

5.3.4 Threats to Validity. Although the results with the evaluated abstract domains are promising, other domains and configurations might require further benchmarks, as CEGAR includes a broad set of possible domains and parameters. While some of these may perform better in some cases, we believe it is unlikely that this effect could compensate for the observed performance degradation with SSA.

Due to the differences in what the frontends can handle, we had to narrow down the subset of input tasks to only around 2000 C programs – this is a definite loss in how representative the measurements are, but still includes 5 distinct subcategories.

The benchmarks were evaluated by using the metric of successfully solved tasks in 15 minutes – the same metric, which is used on SV-COMP to calculate a tool's score. On the other hand, we did not evaluate the results based on CPU or wall time, as based on earlier and current results, most tasks that the tool is capable to solve are solved in under a few minutes, making it hard to draw any conclusions based on this metric.

5.4 Reproducibility

An artifact for the reproduction of these experiments is publicly available at Zenodo [5]. This includes the sources and binaries of the implementation, as well as the C programs used in the evaluation process.

6 CONCLUSION AND FUTURE WORK

In this paper, we intended to shed light on the less obvious difficulties of implementing software model checkers for programming languages (in this case, C): the parsing and processing of the input program. While it is very appealing to use mature compiler frameworks as frontends for the verification algorithms, we showed that existing tools exhibit a bias towards certain combinations, favoring well-established technologies implemented in the LLVM framework in some cases and avoiding them in others. We highlighted that this bias is probably due to the fact that the LLVM framework transforms programs into an intermediate representation that comes in a static single assignment form. We presented our arguments on why this is disadvantageous for abstraction-based algorithms, which is based on theoretical reasons as well as our own experience. Then we outlined an ideal frontend that would combine the benefits of existing technologies, but tailored for formal verification instead of the generation of executables. To demonstrate that CEGAR-based algorithms are indeed highly sensitive to the SSA form, we implemented a custom frontend along the lines of the ideal solution beside the former LLVM frontend of our model checker. Our performance evaluation clearly shows that even with such a low effort (compared to the development that went into LLVM) we can achieve much higher performance – but custom approaches will be unlikely to reach the coverage of LLVM in terms of language support. Therefore, we hope that this paper will inspire discussion in the community and potentially motivate the design and implementation of an LLVM-like framework tailored for formal verification tools.

ACKNOWLEDGMENTS

This research was partially funded by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU grant agreement No. 826452 (https://cordis.europa.eu/project/id/826452), NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003); and by the ÚNKP-21-2-I-BME-142 New National Excellence Program of the Ministry for Innovation and Technology from the Source of the National Research, Development and Innovation Fund.

REFERENCES

[1] Zsófia Ádám, Gyula Sallai, and Ákos Hajdu. 2021. Gazer-Theta: LLVM-based Verifier Portfolio with BMC/CEGAR (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 433–437.

[2] Pavel Andrianov, Karlheinz Friedberger, Mikhail Mandrykin, Vadim Mutilin, and Anton Volkov. 2017. CPA-BAM-BnB: Block-Abstraction Memoization and Region-Based Memory Models for Predicate Abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 355–359.

[3] P. S. Andrianov. 2020. Analysis of Correct Synchronization of Operating System Components. *Programming and Computer Software* 46, 8 (01 Dec 2020), 712–730. https://doi.org/10.1134/S0361768820080022

[4] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. 2013. Domain Types: Abstract-Domain Selection Based on Variable Usage. In *Hardware and Software: Verification and Testing*. Springer International Publishing, 262–278. https://doi.org/10.1007/978-3-319-03077-7_18

[5] Levente Bajczi, Zsófia Ádám, and Vince Molnár. 2022. C for Yourself: Comparison of Front-End Techniques for Formal Verification (Artifact Submission). https://doi.org/10.5281/ZENODO.5920382

[6] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. 2017. Model Checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer International Publishing, Cham, 201–207.

[7] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387.

[8] Dirk Beyer. 2021. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12652)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 401–422. https://doi.org/10.1007/978-3-030-72013-1_24

[9] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 25–32. https://doi.org/10.1109/FMCAD.2009.5351147

[10] Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2015. Combining k-Induction with Continuously-Refined Invariants. *CoRR* abs/1502.00096 (2015). arXiv:1502.00096 http://arxiv.org/abs/1502.00096

[11] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (Sept. 2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[12] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. *Computer Aided Verification* (2007), 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[13] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–190.

[14] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2017. Reliable benchmarking: requirements and solutions. 21, 1 (Nov. 2017), 1–29. https://doi.org/10.1007/s10009-017-0469-y

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Advances in Computers* (2003), 117–148. https://doi.org/10.1016/s0065-2458(03)58003-2

[16] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety Verification and Refutation by k-Invariants and k-Induction. In *Static Analysis*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 145–161.

[17] Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5505)*, Stefan Kowalewski and Anna Philippou (Eds.). Springer, 174–177. https://doi.org/10.1007/978-3-642-00768-2_16

[18] Marek Chalupa, Jan Strejček, and Martina Vitovská. 2018. Joint Forces for Memory Safety Checking. In *Model Checking Software*, María del Mar Gallardo and Pedro Merino (Eds.). Springer International Publishing, Cham, 115–132.

[19] Eti Chaudhary and Saurabh Joshi. 2019. Pinaka: Symbolic Execution Meets Incremental Solving. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 234–238.

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50, 5 (Sept. 2003), 752–794. https://doi.org/10.1145/876638.876643

[21] Edmund M. Clarke, Orna Grumberg, and Doron Peled. 1999. *Model checking*. MIT Press.

[22] Lucas Cordeiro and Bernd Fischer. 2011. Verifying Multi-Threaded Software Using Smt-Based Context-Bounded Model Checking. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 331–340. https://doi.org/10.1145/1985793.1985839

[23] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247.

[24] Yulia Demyanova, Helmut Veith, and Florian Zuleger. 2013. On the concept of variable roles and its use in software analysis. In *2013 Formal Methods in Computer-Aided Design*. IEEE. https://doi.org/10.1109/fmcad.2013.6679414

[25] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software verification using K-Induction. *Static Analysis* (2011), 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[26] Evren Ermis, Jochen Hoenicke, and Andreas Podelski. 2012. Splitting via Interpolants. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–201.

[27] Mikhail Y. R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro. 2017. Handling loops in bounded model checking of C programs via k-induction. *International Journal on Software Tools for Technology Transfer* 19, 1 (01 Feb 2017), 97–114. https://doi.org/10.1007/s10009-015-0407-9

[28] Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 355–365.

[29] Marius Greitschus, Daniel Dietsch, and Andreas Podelski. 2017. Loop Invariants from Counterexamples. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 128–147.

[30] Ákos Hajdu and Zoltán Micskei. 2020. Efficient Strategies for CEGAR-Based Model Checking. *J. Autom. Reason.* 64, 6 (2020), 1051–1091. https://doi.org/10.1007/s10817-019-09535-x

[31] Ákos Hajdu and Zoltán Micskei. 2020. Efficient Strategies for CEGAR-based Model Checking. *Journal of Automated Reasoning* 64 (2020), 1051–1091. https://doi.org/10.1007/s10817-019-09535-x

[32] Les Hatton. 2004. Safer language subsets: an overview and a case history, MISRA C. *Inf. Softw. Technol.* 46, 7 (2004), 465–472. https://doi.org/10.1016/j.infsof.2003.09.016

[33] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–52.

[34] ISO/IEC 9899:201x 2010. *Programming languages — C.* International Standard. International Organization for Standardization, International Electrotechnical Commission.

[35] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391.

[36] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[37] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 425–436. https://doi.org/10.1145/1993498.1993548

[38] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 8559. Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[39] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 12–27. https://doi.org/10.1145/73560.73562

[40] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: a Framework for Abstraction Refinement-Based Model Checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, Daryl Stewart and Georg Weissenbacher (Eds.). 176–179. https://doi.org/10.23919/FMCAD.2017.8102257