Towards Evaluating Size Reduction Techniques for Software Model Checking

Gyula Sallai[1] Ákos Hajdu[1,2]* Tamás Tóth[1]† Zoltán Micskei[1]

[1]Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary

[2]MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary

salla@sch.bme.hu, {hajdua,totht,micskeiz}@mit.bme.hu

Formal verification techniques are widely used for detecting design flaws in software systems. Formal verification can be done by transforming an already implemented source code to a formal model and attempting to prove certain properties of the model (e.g. that no erroneous state can occur during execution). Unfortunately, transformations from source code to a formal model often yield large and complex models, making the verification process inefficient and costly. In order to reduce the size of the resulting model, optimization transformations can be used. Such optimizations include common algorithms known from compiler design and different program slicing techniques. Our paper describes a framework for transforming C programs to a formal model, enhanced by various optimizations for size reduction. We evaluate and compare several optimization algorithms regarding their effect on the size of the model and the efficiency of the verification. Results show that different optimizations are more suitable for certain models, justifying the need for a framework that includes several algorithms.

Keywords: size reduction, compiler optimizations, slicing, model checking, CEGAR

1 Introduction

As our reliance upon safety-critical computer systems grows, so does our natural desire for reliable proofs of their fault-free behavior. Such proofs can be given by formal verification algorithms, such as *model checking*. Unlike testing, these are not only able to prove the presence of errors, but their absence as well, thus they are able to give a satisfactory answer on the safety of a system.

While there are several formal verification techniques available, incorporating them into a development workflow can pose a challenge. In model-driven development, we first design a model that describes our system, then we use formal verification to prove its safety and finally we create an implementation based on the defined (and safe) model. However, designing and defining a model for a project can be rather difficult and in many cases the financial and time constraints do not permit it.

A possible solution to overcome this problem is to transform the already implemented source code to a formal model. However, a drawback of this approach is the large size of the model generated from the source code. As most verification algorithms have a rather demanding computational complexity (usually operating in exponential time and beyond), the resulting model may not admit efficient verification. A way to resolve this issue is to reduce the size of the generated model using *optimizing transformations*.

[*] Partially supported by Nemzeti Tehetség Program, Nemzet Fiatal Tehetségeiért Ösztöndíj 2016 (NTP-NFTÖ-16).

[†] Partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

A. Lisitsa, A. Nemytykh, M. Proietti (Eds.):
Verification and Program Transformation (VPT 2017)
EPTCS 253, 2017, pp. 75–91, doi:10.4204/EPTCS.253.7

This paper describes a framework which implements a *transformation workflow* from C programs to a formal model, known as control flow automaton. This process is enhanced by some common optimization transformations, usually known from compilers. These simplify the model and reduce its size. Then the model is split into several smaller, more easily verifiable chunks using *program slicing* [24]. As a result, the verification algorithm has to handle several smaller problems instead of a large one.

In this paper, we evaluate the effects of such transformations on the size of the model and the efficiency of verification. We use the compiler optimization algorithms known as *constant propagation*, *constant folding* and *dead branch elimination* [1]. We also present and compare different program slicing techniques, namely *backward slicing* [14], *thin slicing* [22] and *value slicing* [18]. The models obtained after these transformation passes are then verified using *predicate abstraction* in conjunction with the *counterexample-guided abstraction refinement* (CEGAR) model checking algorithm. Our results show that the optimizations may allow significant reduction in the size of the model and the execution time of the verifier algorithm. However, the relative effectiveness of the algorithms varies between different input models, which justifies the need for a configurable framework that supports several optimization and slicing algorithms.

The rest of this section discusses related work. Section 2 introduces the preliminaries of our work. Section 3 presents our transformation workflow from C programs to formal verification along with some implementation details. Section 4 evaluates the transformations and their effect on verification. Finally, Section 5 concludes our work.

Related work. Predicate abstraction [15], used in conjunction with CEGAR [11] is a widely used technique for model checking software [2, 6, 13, 10, 7]. However, the performance of these algorithms greatly depends on the size and complexity of the input model and source-to-model transformations tend to produce large models. Several size reduction methods exist, the most prominent of them is program slicing [24, 14]. Several other variants of slicing have been proposed since then, including thin [22] and value slicing [18]. The software model checking tool SLAB [10] successfully incorporates slicing into its workflow, allowing slicing to be performed on the abstract states. In our tool we handle slicing and verification separately and use the former as a preprocessing step. There has also been work on evaluating and comparing software model checkers [5, 8]. However, these papers target the model checking algorithms, whereas the primary focus our work is on the size reduction techniques.

2 Background

In this section we give a brief introduction to the theoretical background of formal verification, program representations, dependency structures and some compiler optimization algorithms used in our work.

2.1 Formal Verification

Model checking is a formal verification technique of mathematically proving correctness or faultiness of computer programs by systematically exploring their state space. There are several program representations suitable for model checking. In this paper we use a formalism known as *control flow automaton* (CFA) [4]. A CFA is a 4-tuple (L, E, ℓ_0, ℓ_e), where $L = \{\ell_0, \ell_1, \ldots, \ell_n\}$ is a set of locations representing program counter values, $E = L \times Ops \times L$ is a set of directed edges representing possible control flow paths. The edges are labeled with operations in *Ops* which get executed when control jumps from one location to another. The distinguished initial location $\ell_0 \in L$ marks the entry point of the program and the special error location $\ell_e \in L$ embodies an undesirable state in the program. In our work, we use ℓ_e to represent failing assertions in a C program. We focus on *assertion checking*, i.e., showing that ℓ_e cannot be reached in any feasible executions of the program.

CEGAR. A typical drawback of using model checking techniques is their high computational complexity. Abstraction-based methods are often applied to overcome this issue. However, it is not straightforward to find the proper precision of abstraction that is coarse enough to reduce complexity but fine enough to prove the desired property. Counterexample-Guided Abstraction Refinement (CEGAR) is an automatic algorithm that starts with a coarse initial abstraction and refines it based on the counterexamples until the proper precision is reached [11]. CEGAR was first described using transition systems and predicate abstraction, but since then many variants have been developed.

In this work we use our generic CEGAR framework [16] for verification, which incorporates many variants of the algorithm. The framework explores the abstract state space in a given *abstract domain* with a given *exploration strategy*. Currently, predicate [15] and explicit value [12] domains are supported with breadth- and depth-first search exploration strategies. The abstract state space is an overapproximation of the original, therefore if no erroneous abstract state is reachable, the original model is also correct. However, if an abstract counterexample is found, its feasibility in the original model is checked. A feasible counterexample corresponds to a failure in the original model. On the other hand, if the counterexample is infeasible (also called spurious), the abstraction is refined. The framework currently supports refinement based on binary or sequence interpolation [21, 23] and unsat cores [20]. The model is then checked again with the refined abstraction and this procedure is repeated until the model is proved to be safe or a feasible counterexample is found. We first defined our framework for transition systems [16] but since, the algorithms have been adapted to CFAs.

2.2 Program Representations

In this paper, we focus on the optimization and verification of C programs. In order to do this, we transform the textual representation into structures more sufficient for dependency analysis, which are needed for the size reduction transformations.

While the control flow automaton representation of a program is suitable for verification, most analysis and transformation algorithms are defined over the language-agnostic formalism of *control flow graphs* (CFG) [1]. A CFG is a 4-tuple (S, E, s_0, s_q), where $S = \{s_0, s_1, \ldots, s_n\}$ is a set of atomic instructions and $E = \{(s_i, s_j), (s_k, s_l), \ldots\}$ is a set of directed edges representing possible control flow paths. The edge $(s_i, s_j) \in E$ iff there is a conditional or unconditional jump from s_i to s_j in the program. The distinguished $s_0 \in S$ and $s_q \in S$ nodes mark the entry and exit points of the program, respectively. As there is a one-to-one correspondence between CFG nodes and program instructions, we shall use the two terms interchangeably.

During analysis, it is often useful to know whether an instruction writes variables that are later read by another instruction. We say that instruction s *defines* the variable v if s assigns a value to v. A *definition* is a pair $d = (s, v)$ where s is an instruction and v is the variable defined in s. Given a node $t \neq s$, d is a *reaching definition* for t, if there is a control flow path between s and t, which contains no other definition of v. If s has a reaching definition for t, then t is said to be *flow dependent* on s. A common alternative name for flow dependency is *data dependency*.

This flow dependency information can be stored in a structure known as use-define chain (UD-chain for short) [1]. Given a program P with the definitions $D = \{d_1, d_2, \ldots, d_n\}$ and an instruction set $S = \{s_1, s_2, \ldots, s_k\}$, the *use-define chain* of P is a set of pairs $\{(s_1, D_1), (s_2, D_2), \ldots, (s_k, D_k)\}$ where

$D_i \subseteq D$ is the set of definitions reaching s_i for all $1 \leq i \leq k$. Given an instruction s, querying the UD-chain yields all instructions on which s flow depends.

It is also useful to know which instructions decide whether some other instruction gets executed or not. This information can be obtained by analyzing the post-dominance relations of the program [14]. A node s *post-dominates* a node t, if every control flow path between t and s_q (the exit location) contains s. If $s \neq t$, then s *strictly post-dominates* t. A node t is *control dependent* on a node s if there is a path from s to t where all nodes are post-dominated by t and t does not strictly post-dominate s.

In order to represent these dependency relations of a program, we use a structure known as *program dependence graph* (PDG) [14]. A PDG is a triple (S, C, D), where $S = \{s_1, \ldots, s_n\}$ is a set of instructions, C is a set of control dependency edges, and D is a set of data dependency edges. The edge $(s_i, s_j) \in C$ if s_j control depends on s_i. The edge $(s_k, s_l) \in D$ if s_l flow depends on s_k.

2.3 Compiler Optimizations

In order to reduce the resulting model's size and complexity, we use *optimization transformations* usually known from compiler theory [1]. *Constant folding* finds and evaluates constant expressions in the program during compile time. *Constant propagation* substitutes variables having a constant value with their respective constant literal. As constants can be local or global, both information need to be propagated. This can be achieved by querying the use-define information of the program. In some cases, constant propagation and folding are able to replace branching decisions with the literals `true` or `false`. *Dead branch elimination* examines these branches and removes inviable execution paths (e.g. the `true` path of a branch with a condition of the boolean literal `false`).

For easing interprocedural analysis, we also use a technique known as *function inlining*. Function inlining is the procedure of replacing a function call with the callee's body. In our work, we use it to obtain more information on the behavior of an interprocedural program, as without more thorough interprocedural analysis, function calls would act as black boxes. A model checker algorithm may extract more information from the entire inlined function body than from merely just a function call.

2.4 Program Slicing

Program slicing is a size reduction technique that attempts to remove all nodes irrelevant to a given instruction and a given set of variables [24]. Let P be an input program, let V be a subset of its variables, and let s be an instruction in P. A *program slice* P' *of* P *with respect to the criterion* (s, V) is a subprogram of P, which produces the same output and assigns the same values to V as the original program P in its statement s.

Figure 3a presents an example with a simple C code snippet, which calculates the sum of the natural numbers less than 11, and asserts that the loop counter and the calculated sum cannot be zero. A slice of this program with respect to the criterion $(9, \{i\})$ is shown in Figure 3b. As it can be observed, the only statements preserved are those relevant to the criterion instruction.

As we are using verification for detecting failing assertions, we use program slicing to split a larger input program into several smaller chunks, with the criteria being the assertion instructions and the variables they use. This will result in smaller verifiable programs instead of a larger one, more precisely every assertion of the program gets verified independently. A proof of a slice's faultiness also indicates that the original program is faulty. On the other hand, if all slices are safe, it means that no assertion in the original program can fail, meaning that the whole program is also safe.

```
 1  int i = 0;
 2  int sum = 0;
 3
 4  while (i < 11) {
 5      sum = sum + i;
 6      i = i + 1;
 7  }
 8
 9  assert(i != 0);
10  assert(sum != 0);
```

(a) A simple C program snippet.

```
 1  int i = 0;
 2
 3
 4  while (i < 11) {
 5
 6      i = i + 1;
 7  }
 8
 9  assert(i != 0);
10
```

(b) Slice of **(a)** with the criterion of $(9, \{i\})$.

Figure 1: An example on slicing.

Several program slicing techniques exist, depending on the use case (model checking, debugging, test generation, etc.). In our current work, we use three methods, all of which are suitable or especially tailored for model checking.

Backward slicing. The most commonly used approach for slicing is a technique known as *backward slicing*. Backward slicing produces accurate slices, while retaining all instructions which are crucial to the slicing criterion. The size of the resulting slices are roughly around 30% of the size of the original program on average [9]. Given a criterion instruction s, backward slicing finds all nodes on which s data depends transitively. As some branching decisions may affect whether a particular instruction is reachable or not, these also need to be included in the slice. These branching decisions are the same as the control dependencies of a given instruction. Of course, these dependencies can have dependencies which need to be taken into account. Therefore backward slicing is done by retaining all instructions on which the criterion control or flow depends transitively. This information can be queried from a program dependence graph.

A backward slicer algorithm marks all nodes which are backwards reachable (walking backwards on both data and control dependency edges) from the criterion node in the program dependence graph [14]. As the PDG explicitly shows the control and flow dependency relations of every instruction, this method will include all required nodes in the slice. Figure 2 shows an example of this procedure (for the same code as in Figure 1) with the assertion node being the criterion. Solid lines represent control dependency, dashed lines represent flow dependency, and filled nodes are those which are backwards reachable from the criterion node.

While backward slicing is simple and accurate, there are other algorithms that sacrifice faithfulness for even greater program size reduction. Such algorithms can be rather useful in the context of model checking. As discussed above, given a slicing criterion $\gamma = (s, V)$, backward slicing attempts to find all transitive control and flow dependencies of γ. In many cases the control dependencies are only required to keep the structure of the program, their respective branch decisions do not affect γ. This idea serves as the basis of the slicing techniques described below.

Thin slicing. A technique known as *thin slicing* [22] aggressively reduces the program size by retaining flow dependencies only. All control dependencies are replaced with nondeterministic boolean predicates (called *abstract predicates* – we denote such predicates with ϕ). After performing this substitution, the thin slicing algorithm behaves the same as backward slicing (in the sense that we are performing a backward search in the PDG). Abstract predicates have no flow dependencies, therefore no such dependencies

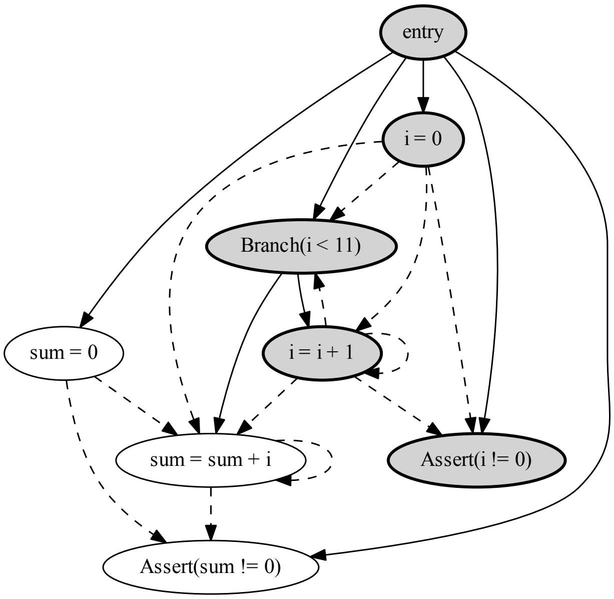


Figure 2: Slicing on the `Assert(i!=0)` node using a program dependence graph.

will be included in the slice, thus allowing a much larger reduction.

On the other hand, abstract predicates may introduce new possible execution paths into the program. In the original program (or a backward slice), a verification algorithm may realize that a certain execution path is inviable due to the value of a branch condition. However, if the program has abstract predicates only, this assumption cannot be made. This also means that the verifier algorithm has to discover all program paths in the resulting slice.

The inclusion of dynamically inviable execution paths also means that verification algorithms may produce more spurious counterexamples on thin slices. Finding whether a counterexample can hold for the original program requires the refinement of the thin slice (which means the inclusion of previously abstracted control dependencies), and running the verifier again on the refined slice. The details of such refinements will be discussed later on.

Value slicing. A possible middle ground between thin and backward slices is called *value slicing* [18]. Value slicing recognizes which control dependencies actually determine the value of the criterion variables and retains those as well. This results in slightly larger program slices compared to thin slicing. However, verification algorithms produce less spurious counterexamples on these slices, thus they require less refinement. Value slicing introduces the term *value-impacting instruction*. Given a slicing criterion γ, an instruction s value-impacts γ if any of the following conditions hold:

1. The slicing criterion γ flow depends on s.
2. There exists an instruction t which value-impacts γ and t flow depends on s.
3. The instruction s is branch decision with conditional jumps to the instructions t_1 and t_2, from which there exist paths $\pi_1 = \{t_1, \dots, \gamma\}$ and $\pi_2 = \{t_2, \dots, \gamma\}$ in the CFG, starting with t_1, t_2 and ending in

the first occurrence of γ. Furthermore, there exists a node $t \neq s$ such that t value-impacts γ and t is the first value-impacting node along π_1 and t is not the first value-impacting node along π_2.

The first and second conditions make sure that the flow dependencies of the remaining instructions are all accounted for. The third condition marks branch decisions which will be retained. A branch decision is value-impacting (and therefore should not be abstracted) if both its outgoing edges are program paths to the slicing criterion and it decides whether a value-impacting instructions gets executed or not.

As mentioned before, value and thin slicing may require refinement in order to prove that a counterexample encountered is not a spurious one. This can be done by selecting an abstract predicate ϕ from the slice and adding it to the slicing criterion, then performing a slicer algorithm again. The selection of such abstract predicates can be done randomly (our current implementation does this) or by using certain heuristics (e.g. distance from the criterion node in the PDG, etc.). Note that the slicing algorithm used during the refinement can be different than the initial slicer. For example, using thin slicing as refinement means that we are refining the slice with a single abstract predicate each time. On the other hand, using value slicing as refinement means that the refined slice may include multiple abstract predicates at once.

```
1  extern int fn1();
2  extern int fn2();
3  extern int fn3(int x, int y);
4
5  int main() {
6      int i = 0, j = 0;
7      int t = fn1();
8      int x = fn3(i, j);
9      int y = 0;
10
11     while (t < 1000) {
12         int s = fn2();
13         if (s == 1) {
14             y = y + x;
15         } else {
16             i = i + 1;
17             j = j + 1;
18         }
19
20         assert(y != 0);
21
22         x = fn3(i, j);
23         t = fn1();
24     }
25
26     return 0;
27 }
```

(a) Backward slice.

```
1  // fn1() removed
2  extern int fn2();
3  extern int fn3(int x, int y);
4
5  int main() {
6      int i = 0, j = 0;
7
8      int x = fn3(i, j);
9      int y = 0;
10
11     while (ϕ) {
12         int s = fn2();
13         if (s == 1) {
14             y = y + x;
15         } else {
16             i = i + 1;
17             j = j + 1;
18         }
19
20         assert(y != 0);
21
22         x = fn3(i, j);
23
24     }
25
26     return 0;
27 }
```

(b) Value slice.

```
1  // fn1() removed
2  // fn2() removed
3  extern int fn3(int x, int y);
4
5  int main() {
6      int i = 0, j = 0;
7
8      int x = fn3(i, j);
9      int y = 0;
10
11     while (ϕ1) {
12
13         if (ϕ2) {
14             y = y + x;
15         } else {
16             i = i + 1;
17             j = j + 1;
18         }
19
20         assert(y != 0);
21
22         x = fn3(i, j);
23
24     }
25
26     return 0;
27 }
```

(c) Thin slice.

Figure 3: An example on backward, value, and thin slices with the criterion $(20, \{y\})$.

Figure 3 offers an example (a simplified version of an example in [18]) on the different slicing methods described above. Suppose `fn1`, `fn2`, and `fn3` are complex operations with multiple different computations and all examples are slices on the criterion $(20, \{y\})$. As it can be seen in Figure 3a, a backward slice retains all instructions which are relevant to the given criterion in any way. The thin slice (shown in Figure 3c) abstracted branch conditions away, by replacing them with ϕ_1, ϕ_2. Note that because of the removed branch conditions in line 11 and 13, the calculation of their used variables is also omitted from the slice. This also allowed the removal of `fn1` and `fn2`. The value slice (Figure 3b) also abstracted the loop condition in line 11 away, but it kept the condition in line 13. This is because of the fact that the value of *y* is actually determined here: line 13 decides whether the value-impacting assignment in line 14 gets executed or not. If the branch condition is true, *y*'s value changes, but if it is false, *y* stays the same. The function `fn2` was retained as well, because line 13 depends on the value of *s*, which was defined in line 12 by using the value returned by `fn2`.

3 Transformation Workflow

In this section we propose a workflow that is able to transform C programs to control flow automata. In order to reduce the size of the resulting models, we enhance this process with compiler optimizations and program slicing. The resulting models can then be verified using an arbitrary verification algorithm. In this paper, we use CEGAR-based algorithms [16] for this purpose.

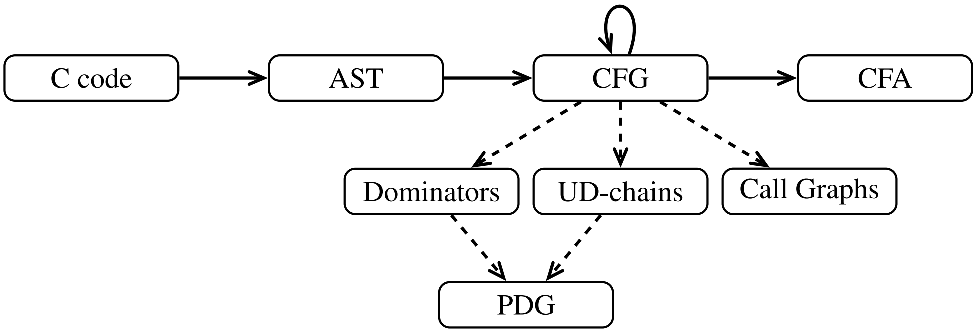


Figure 4: Transformation workflow.

An overview of the transformation workflow is shown in Figure 4. First, we take a C source code file as an input. This source code is then parsed into an *abstract syntax tree* (AST), which describes the syntactic structure of the program. The abstract syntax tree is then transformed into a control flow graph. This CFG is then simplified by the application of compiler optimizations, namely function inlining, constant propagation, constant folding, and dead branch elimination. For further size reduction, we also apply program slicing. Several dependency structures are required for these transformations:

- program dependence graphs for slicing,
- dominator relation information for constructing the PDG,
- *use-define* information (*UD-chains*) for the PDG and also for constant propagation,
- and *call graphs* for function inlining.

After running the optimization passes, we perform slicing on the resulting CFG. This operation splits a single CFG into several smaller ones. Our framework supports all backward, value, and thin slicing. Currently the slicer criteria are the assertion instructions (calls to the `assert` function in C) and their associated variables in the control flow graph, meaning that each assertion gets its own slice. These slices are then transformed into control flow automata. Due to the slicing criteria, error locations in the resulting automata represent failing assertions of the original program.

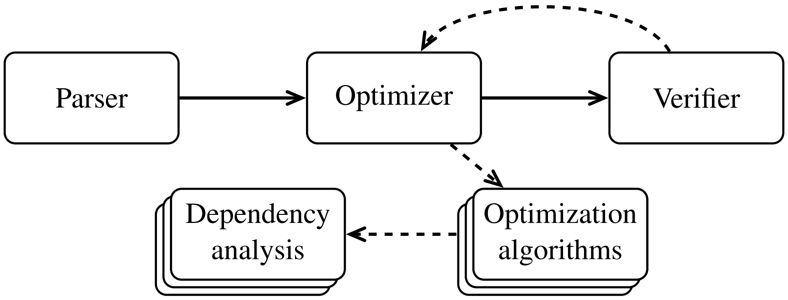


Figure 5: Architecture of the implementation.

Implementation. We implemented a prototype of our workflow in Java. Figure 5 offers an overview of the overall architecture, which builds on three loosely dependent components: parser, optimizer and verifier.

The parser component uses the parsing library of the Eclipse C/C++ Development Tools plug-in (CDT).[1] The CDT parser handles source code lexing and parsing, after which it yields an abstract syntax tree. This AST is then transformed into a control flow graph. At the current state of the project, only a restricted set of C language features is supported. The current implementation only allows the usage of control structures (**if-then-else**, **do-while**, **while-do**, **switch**, **break**, **continue**, **goto**) and non-recursive functions. Types are restricted to integers and Booleans only. Arrays and pointers are not supported at the moment.

The optimizer module performs the optimization transformations. This component is able to run a configurable number of optimization passes. The implementation currently supports constant propagation, constant folding, dead branch elimination, and function inlining. Some optimizations may require certain dependency information of the input program, therefore they need to be able to query this information at any time. The optimizer also handles program slicing. The module is able to use an arbitrary slicing algorithm (which conforms to a given interface per se) to perform slicing. Currently we support the methods presented in the previous sections, namely traditional backward slicing, thin slicing, and value slicing. After the acquisition of the smaller program slices from the slicer algorithm, the optimizer transforms these CFG slices into control flow automata.

These automata are then checked by the verifier component for assertion violations. While this component is modular and replaceable, our current implementation focuses solely on predicate abstraction in conjunction with a CEGAR-based algorithm [16] adapted to CFAs. As refinement may be needed due to the presence of value and thin slicing, it is possible that the optimizer has to run again and refine the slice.

4 Evaluation

In this section, we evaluate and compare the effects of optimizations regarding the size of the model and the efficiency of verification. First we present the programs used and the environment, and then we discuss the results.

[1] `http://www.eclipse.org/cdt/`

4.1 Objects and Environment

We performed our evaluation on programs from the Competition on Software Verification (SV-Comp) [3] repertoire. This competition aims to compare the soundness and performance of software verification tools. In our evaluation we use some of its tasks to compare and evaluate the performance of the different algorithms implemented in our tool. Other tools are currently not considered as we only aim to evaluate the different algorithms within our framework. The verification tasks used in our work can be separated into three main categories, which are described below.

locks This task set consists of small (100-150 LOC) locking mechanisms described with nondeterministic integers and **if-then-else** constructs. This category consists of files, which have several assertion statements within, therefore producing many (10-15) small slices.

eca The ECA (event-condition-action) category describes large (500-600 LOC) event-driven reactive systems. The events are represented with nondeterministic variables. The files in this category are special in a sense that one large problem is already split up into several files, with each file containing a single assertion, hence a single slice.

ssh-simplified These tasks describe large (500-600 LOC) server-client systems. While these systems are rather complex, verifying the server-client communication is not part of this task, such factors are abstracted away with nondeterministic variables. These programs also produce a single slice.

As our framework currently supports slicing on `assert` instructions, we used slightly modified versions of the programs listed above. The original competition specifications considered a program faulty if a particular error function was called. We replaced such calls with an `assert(false)` instruction so that the call of the error function will lead to the special error location.

Variables of the evaluation are listed in Table 1 grouped into three main categories: parameters of the program (input), parameters of the algorithm configuration (input), metrics of the algorithm (output). In the figures configurations are given by the abbreviation of the possible input options. For example, `VFD` denotes value slicing (`V`), without optimizations (`F`, false), with DFS search strategy (`D`). Individual slices are referred to by concatenating the file name and the slice number with a hash sign (e.g., `locks/locks11_true.c#2`)

All measurements were executed on a 64-bit Windows 7 virtual machine with 2 cores (2.50 GHz) and 16 GB RAM. Each slice was tested with the timeout of 3 minutes. To avoid interprocedural analysis, function inlining was applied to every input program, regardless of the configuration. In our experiment, using a thin slicer for refinements was deemed rather unfavorable, because it runs a new iteration for every abstract predicate. In order mitigate this, we use value slicing as the refinement algorithm for both the thin and value slicer.

4.2 Results and Discussion

In our experiment, we evaluated 9 input programs (3 from **locks**, 4 from **eca** and 2 from **ssh**). With slicing, these programs have produced 50 slices. We checked these slices individually and also checked the original programs without slicing. Given that we have 3 slicing algorithms (+1 for no slicing), 2 search strategies and 2 possible values (true or false) regarding compiler optimizations, the total number of configurations is 16. This gives us a total number of $4 \cdot 9 + 12 \cdot 50 = 636$ measurements. For convenience, from this point forward, we refer to any input model (a whole, unsliced program, or a specific slice) as ”slice”. The number of successful executions (no timeout) is 570, the number of safe results is 484.

Table 1: Variables of the experiment.

Category	Name	Type	Description
Input (program)	File	String	Unique name (and path) of the instance.
	Slice No.	Integer	Index of the slice (assertion) being verified. If the program is not sliced with a given configuration, this variable is ignored.
Input (config.)	Slicer	Factor	Slicing algorithm. Possible values: `NONE`, `THIN`, `VALUE`, `BACKWARD`.
	Optimizations	Boolean	Indicates if constant propagation and dead branch elimination transformations are used.
	Search	Factor	Search strategy during verification. Possible values: `BFS`, `DFS` (breadth- and depth-first search).
Output (metrics)	Safe	Boolean	Verification result, indicates whether the given slice was deemed safe or unsafe by the verifier.
	InitLocs	Integer	Initial location count in the CFA corresponding to the slice.
	InitEdges	Integer	Initial edge count in the CFA corresponding to the slice.
	ArgSize	Integer	Number of nodes in the Abstract Reachability Graph (ARG), i.e., the number of explored abstract states.
	EndLocs	Integer	Final location count in the CFA corresponding to the slice.
	EndEdges	Integer	Final edge count in the CFA corresponding to the slice.
	Optimization time	Integer	Execution time of the optimizer component (including optimizations, slicing, and slice refinements), in milliseconds.
	Verification time	Integer	Execution time of the verification algorithm, in milliseconds.

Figure 6 gives a high level overview of the distribution and range of output variables, grouped by task categories. As it can be seen, values for the **eca** and **ssh** task sets are on a rather similar scale. The **locks** set, however, produces significantly smaller numbers and more outliers. Therefore, we present those results separately from the other two categories, where this is required. We also found that there is a strong correlation between the location and edge count of the CFAs (with an R^2 value of 0.998), therefore we only use the former to describe the size of a CFA.

4.2.1 Impact of Slicing and Optimizations on CFA Size

An overview of the size reduction effect of the different slicing methods on the initial and final size of the CFA is shown in Figure 7. It can be seen that all three slicing techniques reduce the size of most programs significantly compared to the case when no slicing is applied. As it can be seen in the left, thin and value slicing allow even greater reduction initially, but the plots on the right show that the size of the final automata (after refinements) becomes roughly the same as in the case of backward slicing.

Figure 8 compares the initial and final size of the CFA for each slice individually. Naturally, no difference can occur in initial and final size if we use no slicing. As backward slicing requires no

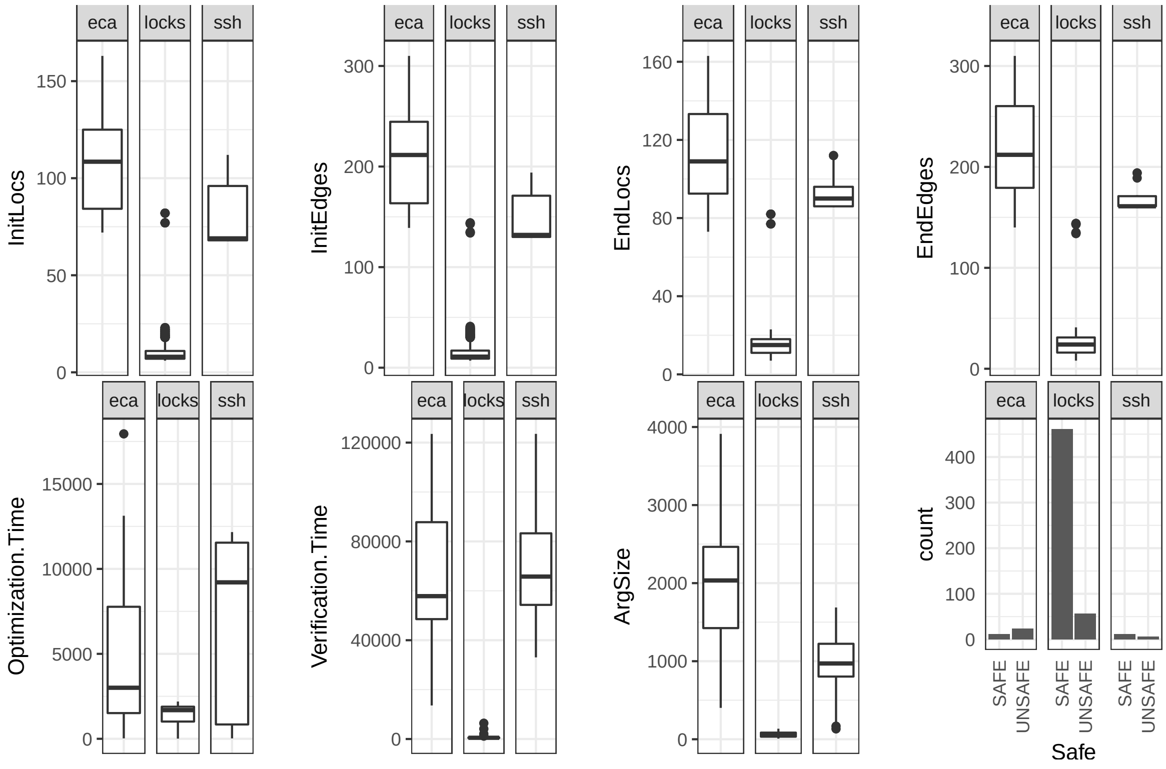


Figure 6: Overview of output variables.

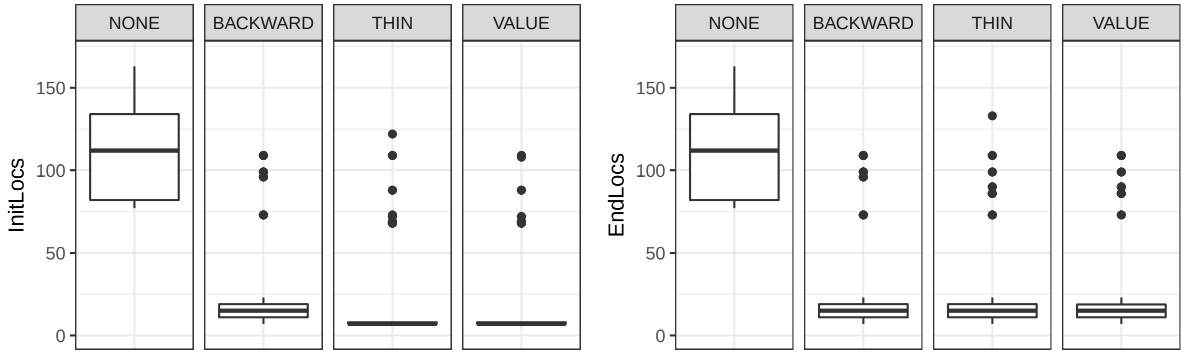


Figure 7: Comparison of the effect of different slicers on the initial and final CFA size.

refinement, size of the backward slices also stays the same. In contrast, value and thin slicing usually requires refinements, which iteratively increase the size of the CFA.

Curiously, the measurements of the value slicer are same as the thin slicer's. This is due to the fact that the value slicer only retains branches that make a choice between the possible values of a variable required by the criterion node. The SV-Comp task set contains programs with a distinctive artificial structure and no such branches are present in the given programs, therefore the heuristics used by the value slicing algorithm could not work in this case.

The heat map of the average initial slice sizes is shown in Figure 9. The vertical axis lists the possible configurations without the search strategy value. For example, NF stands for slicing NONE, optimizations

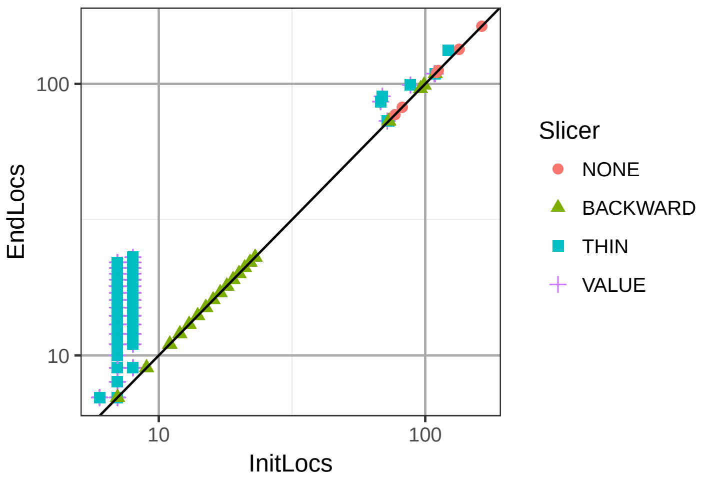
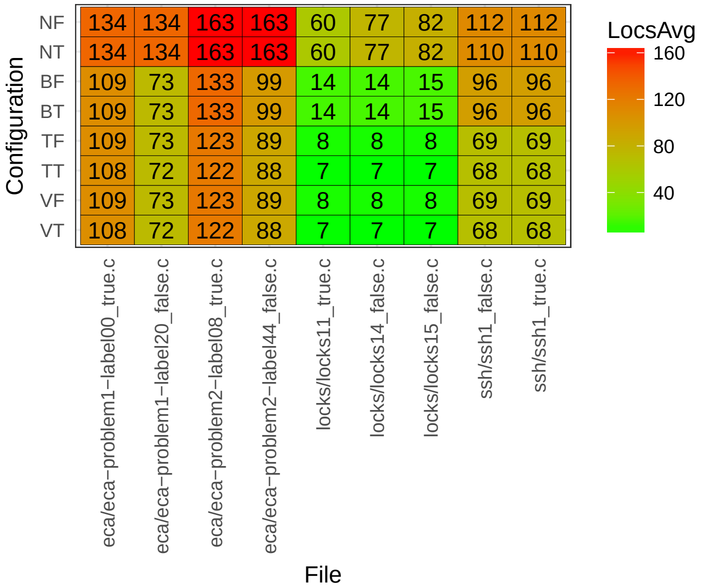


Figure 8: Overview of the initial and final CFA size with different slicers.

FALSE. Our calculations showed that the standard deviation of the slice sizes is 5.13 at most, therefore displaying and evaluating averages has no significant distorting effect. The first row shows CFA sizes without any slicing and optimization algorithms. Backward slicing reduces the CFA size considerably, even more so in the programs of the **locks** category. Value and thin slicing allow even further reductions, except for the **eca** category. Since the assertions in the current programs do not contain any variables, the size of the initial CFAs are the same for both slicers. We can also conclude that compiler optimizations have little to no effect on CFA size. Nevertheless, the simplified expressions built by constant propagation may still allow some decrease in verification execution time.



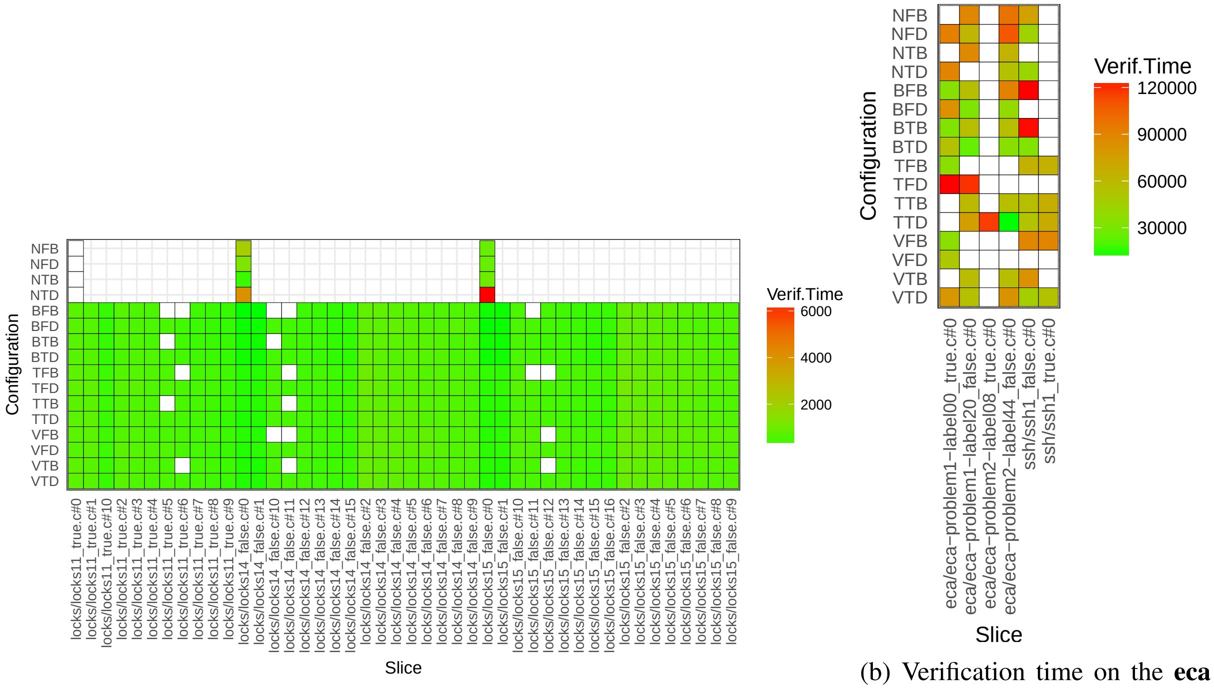
Configuration	eca/eca−problem1−label00_true.c	eca/eca−problem1−label20_false.c	eca/eca−problem2−label08_true.c	eca/eca−problem2−label44_false.c	locks/locks11_true.c	locks/locks14_false.c	locks/locks15_false.c	ssh/ssh1_false.c	ssh/ssh1_true.c
NF	134	134	163	163	60	77	82	112	112
NT	134	134	163	163	60	77	82	110	110
BF	109	73	133	99	14	14	15	96	96
BT	109	73	133	99	14	14	15	96	96
TF	109	73	123	89	8	8	8	69	69
TT	108	72	122	88	7	7	7	68	68
VF	109	73	123	89	8	8	8	69	69
VT	108	72	122	88	7	7	7	68	68

Figure 9: Heat map of average initial slice sizes.

4.2.2 Impact of Slicing and Optimization on Verification Performance

Figure 10 shows a heat map of the verification execution time with different input configurations. White grids represent unsuccessful executions (due to timeouts). The unfilled grids with light gray borders represent cases where a particular measurement data does not exist, because its corresponding slice has not been produced by the given slicer setting. Figure 10a depicts results on the **locks** task set. As it can be seen, the slices in this category mostly produce fairly similar execution times with different slicers. Without slicing however, the verification execution time is quite large: for example, the `locks11_true.c` ran into a timeout with all configurations that did not use slicing. For sliced programs, the verification time is rather promising. Timeouts on some slices with the BFS search strategy suggests that this search method can be inferior to the DFS strategy in a few cases.

The measurements for the **ssh** and **eca** categories (presented in Figure 10b) show really diverse results. In most cases, slicing and optimizations decreased verifier execution time and allowed the verification of programs which have ran into timeout previously. However, actual times vary through different slicing methods and search strategies. For example, the `eca-problem1-label00_true.c` task performed better with the BFS search strategy in most cases, while `eca-problem1-label20_false.c` performed better with DFS. This supports the need for a configurable verification framework – like the one presented in this paper, that is able to support these different slicing methods and search strategies. It is also interesting that there was a task (`eca-problem2-label08_true.c`) that could only be verified using a single configuration (`TTD`, i.e., thin slicing with optimizations, DFS search). This case could be the subject of a more detailed analysis in the future.

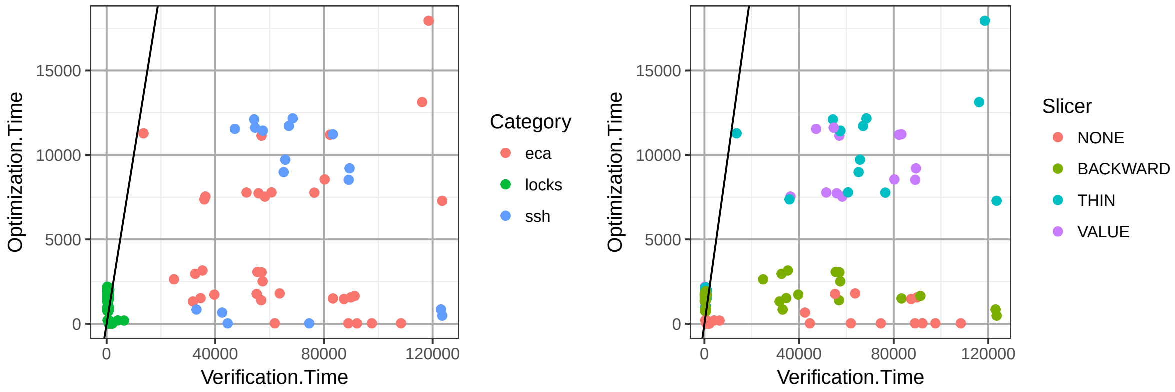


(a) Verification time on the **locks** set.

(b) Verification time on the **eca** and **ssh** sets.

Figure 10: Heat map of verification time for slices with different configurations.

We also compared verification time and optimization time. Their comparison, grouped by program categories is shown in Figure 11a. As it can be seen, optimization takes roughly as much time as verification on small programs (from **locks**). For larger programs however, optimization time is negligible compared to the verifier's execution time. Note that optimization time also includes all time spent on slicing and slice refinement. Figure 11b shows the same comparison grouped by slicer types. We can conclude that without slicing or with backward slicing, the optimization time is rather fast, especially when compared to verification time. For value slicing, the optimization time increases significantly due to the refinements. For thin slicing this time increases slightly further, except for a few exceptions, where thin slicing produced much greater optimization times.



(a) Verification and optimization time by categories.

(b) Verification and optimization time by slicers.

Figure 11: Comparison of verification time and optimization time.

Threats to validity. External validity of our results is currently limited to our tool and the selected categories of SV-COMP. Selecting more models and tools from different sources could improve external validity. Internal validity was increased by a fully automated measurement environment but it could also be improved by for example repeating the same measurements multiple times.

5 Conclusions

In this paper, we have presented and evaluated a framework for transforming C programs to control flow automata, enhanced by optimization transformations known from compiler design and different program slicing methods (namely backward, thin, and value slicing). The resulting models are verified using the CEGAR model checking algorithm. We also performed an experiment by evaluating the performance of the different slicing methods and verification configurations. Our results show that the effectiveness of a certain slicing algorithm varies between certain input programs. We also concluded that in some cases a certain slicing method performs better than the others. All this information supports the need for a configurable framework – like the one presented in this paper, which includes several slicing and optimization algorithms.

Future work. Our framework has several opportunities for improvement. The range of supported features of the C language could be extended with for example arrays, pointers or structs. Other slicing algorithms could be included in the workflow, such as interprocedural slicing [17]. Adding support for the LLVM IR [19] would extend the range of supported languages and would also implicitly add multiple fine-tuned optimizations into the workflow. Moreover, the effects of size reduction could be evaluated on a wider range of verifier configurations, e.g. using different abstract domains or refinement strategies.

References

[1] Alfred V. Aho, Ravi Sethi & Jeffrey D. Ullman (1986): *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] Thomas Ball & Sriram K. Rajamani (2002): *The SLAM Project: Debugging System Software via Static Analysis*. SIGPLAN Not. 37(1), pp. 1–3, doi:10.1145/565816.503274.

[3] Dirk Beyer (2016): *Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)*. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 9636, Springer, pp. 887–904, doi:10.1007/978-3-662-49674-9_55.

[4] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu & Roberto Sebastiani (2009): *Software model checking via large-block encoding*. In: Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design, IEEE, pp. 25–32, doi:10.1109/FMCAD.2009.5351147.

[5] Dirk Beyer & Matthias Dangl (2016): *SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms*. In: Verified Software. Theories, Tools, and Experiments, Lecture Notes in Computer Science 9971, Springer, pp. 181–198, doi:10.1007/978-3-319-48869-1_14.

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

[7] Dirk Beyer, Thomas A. Henzinger & Grégory Théoduloz (2007): *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. In: Computer Aided Verification, Lecture Notes in Computer Science 4590, Springer, pp. 504–518, doi:10.1007/978-3-540-73368-3_51.

[8] Dirk Beyer, Stefan Löwe & Philipp Wendler (2015): *Refinement Selection*. In: Model Checking Software: 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings, Lecture Notes in Computer Science 9232, Springer, pp. 20–38, doi:10.1007/978-3-319-23404-5_3.

[9] David Binkley, Nicolas Gold & Mark Harman (2007): *An Empirical Study of Static Program Slice Size*. ACM Transactions on Software Engineering and Methodology 16(2), doi:10.1145/1217295.1217297.

[10] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner & Heike Wehrheim (2008): *Slicing Abstractions*. Fundamenta Informaticae 89(4), pp. 369–392.

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

[12] Edmund Clarke, Anubhav Gupta & Ofer Strichman (2004): *SAT-based counterexample-guided abstraction refinement*. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 23(7), pp. 1113–1123, doi:10.1109/TCAD.2004.829807.

[13] Edmund Clarke, Daniel Kroening, Natasha Sharygina & Karen Yorav (2005): *SATABS: SAT-Based Predicate Abstraction for ANSI-C*. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 3440, Springer, pp. 570–574, doi:10.1007/978-3-540-31980-1_40.

[14] Jeanne Ferrante, Karl J. Ottenstein & Joe D. Warren (1987): *The Program Dependence Graph and Its Use in Optimization*. ACM Trans. Program. Lang. Syst. 9(3), pp. 319–349, doi:10.1145/24039.24041.

[15] Susanne Graf & Hassen Saidi (1997): *Construction of abstract state graphs with PVS*. In: Computer Aided Verification, Lecture Notes in Computer Science 1254, Springer, pp. 72–83, doi:10.1007/3-540-63166-6_10.

[16] Ákos Hajdu, Tamás Tóth, András Vörös & István Majzik (2016): *A Configurable CEGAR Framework with Interpolation-based Refinements*. In: Formal Techniques for Distributed Objects, Components and Systems, Lecture Notes in Computer Science 9688, Springer, pp. 158–174, doi:10.1007/978-3-319-39570-8_11.

[17] Susan Horwitz, Thomas Reps & David Binkley (1990): *Interprocedural Slicing Using Dependence Graphs*. *ACM Transactions on Programming Languages and Systems* 12(1), pp. 26–60, doi:10.1145/77606.77608.

[18] Shrawan Kumar, Amitabha Sanyal & Uday P. Khedker (2015): *Value Slice: A New Slicing Concept for Scalable Property Checking*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 9035, Springer, pp. 101–115, doi:10.1007/978-3-662-46681-0_7.

[19] Chris Lattner & Vikram Adve (2004): *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, IEEE, pp. 75–86.

[20] Martin Leucker, Grigory Markin & Martin R. Neuhäußer (2015): *A New Refinement Strategy for CEGAR-Based Industrial Model Checking*. In: *Hardware and Software: Verification and Testing*, *Lecture Notes in Computer Science* 9434, Springer, pp. 155–170, doi:10.1007/978-3-319-26287-1_10.

[21] Kenneth L. McMillan (2005): *Applications of Craig Interpolants in Model Checking*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 3440, Springer, pp. 1–12, doi:10.1007/978-3-540-31980-1_1.

[22] Manu Sridharan, Stephen J. Fink & Rastislav Bodik (2007): *Thin Slicing*. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, pp. 112–122.

[23] Yakir Vizel & Orna Grumberg (2009): *Interpolation-sequence based model checking*. In: *Proceedings of the 2009 Conference on Formal Methods in Computer-Aided Design*, IEEE, pp. 1–8, doi:10.1109/FMCAD.2009.5351148.

[24] Mark Weiser (1981): *Program Slicing*. In: *Proceedings of the 5th International Conference on Software Engineering*, IEEE, pp. 439–449.