Master's Thesis

Configurable Software Verification based on Slicing Abstractions

Martin Spießl

2018-06-26

Software and Computational Systems Lab
Department of Informatics
LMU Munich

Assessor: Prof. Dr. Dirk Beyer

Advisor: Dr. Philipp Wendler

Abstract

Abstraction Slicing is a CEGAR-based software verification technique that is used in different contexts by the software model checkers SLAB and Ultimate Kojak. Based on the predicates gained by an spurious counterexample, states in the abstract model are split and infeasible edges removed. Other CEGAR-based approaches like predicate abstraction and lazy abstraction with interpolants remove infeasible states instead of checking edges for infeasibility.

In this thesis, basic versions of SLAB and Kojak are implemented in the CPAchecker framework. This allows to compare them to each other as well to other similar approaches like predicate abstraction or lazy abstraction with interpolants. Optimizations for abstraction slicing are presented that are similar to adjustable-block encoding, thus making it possible to observe the effect of different block sizes on the performance of the new analyses. The evaluation shows that for single-block encoding the new implementation of Kojak can be faster than predicate abstraction. For larger block sizes the overhead of slicing edges instead of just checking states becomes dominant and predicate abstraction becomes faster.

Contents

List of Figures

List of Algorithms

List of Tables

1 Introduction

1.1 Motivation

Computers have an ever increasing impact on our daily lives. From economics, medicine and engineering to the way we socially interact. So when a computer program does not do what it is supposed to do, this can have a wide range of implications. Sometimes it might just be annoying, but it might also cost a lot of money or worse, even lives might be at risk. Just imagine a medical imaging device that fails in the middle of surgery due to a unexpected bug in the software it is running. This explains why a lot of effort is put into ways to write software that works as expected.

One standard technique that is nowadays used in industry is testing, where the program is run multiple times with different input in order to increase the chance of finding bugs before the program is run in the production environment. While testing is an indispensable and useful method, it can almost never give complete certainty that all bugs have been found.

For applications where this certainty is desired, methods of formal verification can be used. Here, one tries to find a mathematical proof that the software fulfills a set of formally specified properties. This can happen manually, interactively with the help of a computer, or even automatically.

Over the past two decades, a wide range of automatic verification techniques have been presented. These have to be compared on a theoretical level in order to find conceptual similarities and differences. On a practical level, their implementations need to be compared in order to find out which technique has the best performance. Both of these goals can be difficult to achieve. When it comes to theory, a common concept can be obscured by the usage of different formalisms. When it comes to empirical results, differences in performance might stem from other factors such as usage of different libraries or programming languages.

It is therefore desirable to have a tool that implements a wide range of techniques and makes them comparable by just changing the blocks that really need to differ. For that, a theoretical framework is needed that can unify different approaches. A way to achieve this is the concept of configurable program analysis (CPA) [1]. The software model checker CPAchecker uses this framework to implement a number of different verification algorithms. Comparing them using CPAchecker can give interesting insights into performance differences and their reasons [2].

The objective of this thesis is to add a specific class of verification techniques, namely the concept of abstraction slicing, to the set of algorithms that can be expressed in the CPA framework. This allows to find further conceptual similarities to existing approaches and also to better understand the influence of certain components on the overall performance.

1.2 Related Work

While deductive model checking [3] uses slicing and splitting of abstractions for verification of infinite state systems, this approach does require some user interaction in order to select the right predicates for splitting.

The slicing-abstractions approach [4] automates this step by using Craig interpolation in a CEGAR loop. The resulting abstraction graph can be seen as correctness certificate. The approach is tailored for general concurrent systems because there is no explicit program counter and optimizations like partial-order reduction aim to mitigate state-space explosion. The algorithm contains many optimizations, e. g., the number of abstract states can be reduced by an optimization called state bypassing. Slicing abstractions is implemented in the software model checker SLAB [5].

A similar approach models the program counter explicitly [6]. This allows to make direct use of the information encoded in the control flow automaton but potentially increases the number of abstract states. Large-block encoding is used as an optimization in order to counteract this effect. The implementation of this algorithm is known as Ultimate Kojak [7].

There is also a relatively new verifier called THETA whose architecture is based around splitting of abstract states [8, 9]. It is however not using abstraction slicing and is therefore much closer related to predicate abstraction or lazy abstraction with interpolants.

Adjustable-block encoding [10] is a generalization of large-block encoding. Instead of transforming the CFA, ABE decides for each new abstract state whether it represents the end of a block using the so-called block operator.

In this work, the algorithms behind Kojak and SLAB are expressed using the CPA framework. For Kojak ABE is used instead of LBE. For SLAB, state bypassing is realized using a new approach called flexible-block encoding that works similar to ABE, but allows to change whether an abstract state is at the block end afterwards.

2 Background

In order to express abstraction slicing using the CPA framework, we ne to get an overview of the different approaches that are also formulated using varying terminology and definitions.

First we need to establish a mathematical notion of the term program. Programs are normally written in high-level programming language and then executed on a computer, which is a finite approximation of a Turing machine. During program execution, the computer will take a series of actions, each changing its internal state, which will be called *concrete state* from now on.

We will start by defining a labeled transition system. Later we define a notion of control flow for these transition systems, which will then lead to our definition of a program.

2.1 Labeled Transition Systems

A *labeled transition system* L is a 4-tuple $L = (T, C, C_0, \rightarrow)$. T is a set of labels (actions). $C \subseteq X \rightarrow \mathbb{V}$ is the set of concrete states which are modeled as mapping from a set X of variables to the set of possible concrete values $\mathbb{V}$. The initial states are given by a set $C_0 \subseteq C$. The transitions are determined by the *transfer relation* $\rightarrow \in C \times T \times C$. The transfer relation can be split into a relation for every action $\xrightarrow{\tau} \subseteq C \times C$. For $c_1, c_2 \in C$ we will write $c_1 \xrightarrow{\tau} c_2$ if $(c_1, \tau, c_2) \in \rightarrow$ and $c_1 \rightarrow c_2$ if there exists a τ such that $(c_1, \tau, c_2) \in \rightarrow$. Note that a transfer relation like $\rightarrow \in C \times T \times C$

can also always be expressed as a *transfer function* $f_{\rightarrow} : C \times T \rightarrow \mathcal{P}(C)$ that maps each element of C to the set of its successor under a certain action in T. $\xrightarrow{\tau}$ has the corresponding function $f_{\xrightarrow{\tau}} : C \rightarrow \mathcal{P}(C)$.

The transfer relation for a program can be expressed with predicates of first-order logic that range over sets X and X' of unprimed and primed variables, representing the state before and after the transition. This means that for every variable x in X there exists a corresponding x' in X'. We will write $\Phi_{\rightarrow}(X, X')$ for the predicate corresponding to $\rightarrow$, and $\Phi_{\tau}(X, X')$ for the predicates corresponding to the individual $\xrightarrow{\tau}$. A predicate $\Phi_{\tau}(X, X')$ over sets X and X' of primed and unprimed variables will be called a *transition formula*. The variables in the predicates can be substituted, e. g. $\Phi[X''/X]$ denotes the predicate where all variables x in X are replaced by their corresponding variable x'' in X''. Likewise $\Phi[c(X)/X]$ denotes the predicate where all variables x in X are replaced by the corresponding value of $c(x)$. The transfer relation and these predicates are equivalent in the following sense:

$$\rightarrow = \{(c_1, \tau, c_2) \mid \tau \in T \wedge \Phi_{\rightarrow}[c_1(X)/X, c_2(X')/X']\}$$

Transition formulas can be sequentially composed in order to express consecutive transitions in a *path formula*:

$$(\Phi_1 \circ \Phi_2)(X, X') = \exists X'' : \Phi_1[X''/X'] \wedge \Phi_2[X''/X]$$

2.2 Control Flow

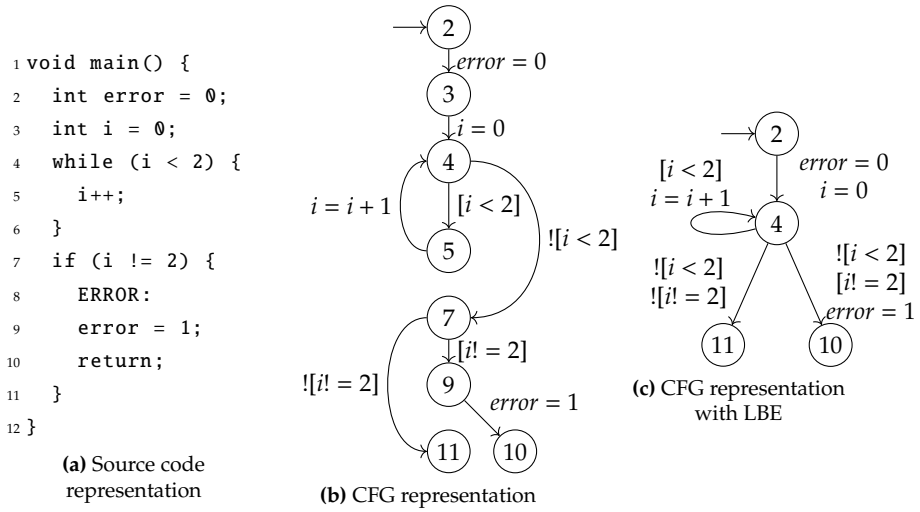
In an imperative programming language, a program is usually given in the representation of its source code. An example is shown in Fig. 1a. The source code implies a certain control flow, i. e., the order in which the statements are traversed. This can be visualized by a control flow graph (CFG), as can be seen in Fig. 1b. Here the nodes are labeled with integers, the program locations.

We can describe this control flow by demanding that the transfer predicates have a certain form that models the program locations with a special variable, the program counter *pc*. This means the set of variables has the form $X = V \cup \{pc\}$ where V is the set of variables that correspond to the concrete data state. All initial states from C_0 shall assign the same value for the program counter. The transfer predicate for each action shall have the following form:

$$\Phi_{\tau}(X, X') \equiv pc' = l' \wedge pc = l \wedge \varphi_{\tau}(V, V')$$

Here $\varphi_{\tau}(V, V')$ is a special transition formula over primed and unprimed data variables which we will call the *data transition formula*. These data transition formulas can also be composed into *data path formulas* $(\varphi_0 \circ \varphi_1 \circ \cdots \circ \varphi_n)(V, V')$. Every data transition formula for itself is already a data path formula. l and l' are values of the program counter before and after the transition. The notion $l \xrightarrow{\tau} l'$ can be used to indicate that the transition τ transfers the control flow from l to l'.

For a simple imperative programming language in which each statement is either an assignment of a single variable or an assumption, the data transition formulas will have a simple structure, either $\varphi_{\tau}(V, V') \equiv (v' = h(V))$ for some



(a) Source code representation

(b) CFG representation

(c) CFG representation with LBE

Figure 1: Different representations of the example program. Note that line numbers in the source code match the node labels in the CFG.

function $h(V)$ and variable $v' \in V'$ or $\varphi_\tau(V, V') \equiv g(V)$ for some predicate $g(V)$. It is sufficient to look at programs that have this structure because more complex statements can be decomposed into several of these simple statements. The example program in Fig. 1 is of such a form. In the control flow graph, the transitions are given in a C-like syntax. Assumptions are given in brackets, assignments without brackets. For example, the assignment "$i = i + 1$" is equivalent to the transition formula "$i' = i + 1$" and the assertion $[i < 2]$ is equal to the transition formula $i < 2$.

2.3 Concrete paths

Each program defines a set of possible concrete paths of execution. A *concrete path* is a sequence $(c_0, \tau_0, c_1), (c_1, \tau_1, c_2), \ldots\ldots, (c_{n-1}\tau_{n-1}, c_n)$ of triples from the transition relation $\rightarrow$, where c_0 is a state from the set C_0 of initial states. Note that it is not enough to just give the concrete states or just give the actions in order to identify a concrete path uniquely, as there might be multiple actions that transfer from a c_i to a c_{i+1} and there might also be multiple pairs of concrete states for the same action.

If just a sequence $\tau_0, \tau_1, \ldots, \tau_n$ of actions is given, this corresponds to a path formula $(\Phi_0 \circ \Phi_1 \circ \cdots \circ \Phi_n)\,(X, X')$ that will be satisfiable if there exists a concrete path with the same sequence of actions (the converse does not hold).

2.4 Specification

All the possible concrete paths of a program can be divided into those that exhibit wanted (good) behavior and those that exhibit unwanted (erroneous) behavior.

A (formal) *specification* is a way to determine for a concrete path whether it is acceptable or not. An example would be that a certain variable, e. g. $err \in X$, may never be set to a value different than 0. In this case, a certain subset of concrete states becomes error states, namely all $c \in C$ that fulfill $c(err) \neq 0$.

Note that a specification does not need to be limited to properties of a single state. In general, it can be given by an arbitrary formula of temporal logic. For simplicity however, we will only look at specifications where certain concrete states can be identified as concrete error states. The specification can then be given as a predicate over the program variables. In the previous example, this would be $error(X) \equiv err \neq 0$. This predicate is what determines whether a concrete state is a *concrete error state*. The problem of finding out whether a program fulfills a formal specification is known as *model checking*.

A *concrete error path* in this framework is a concrete path whose last state c_n fulfills the error predicate (and is therefore a concrete error state): $error[c_n(X)/X] \equiv \top$. These are the concrete paths that exhibit unwanted behavior and thus lead to a specification violation.

2.5 Large-Block Encoding

It is logical that control flow graphs for programs that use such a simple, imperative language are larger than those that allow for more complex instructions. For an analysis this can come at a performance cost. In this case the CFG can be transformed into one where all edges that do not form a loop are composed together, which is known as *large-block encoding*(LBE) [11]. Points at which branchings merge are modeled by disjunction of the path formulas. This means that one transition in LBE can model different paths through the program at once.

In the notion of program that is used here, the result of LBE is a different transition system and as such a different program. There is however a equivalence relation between the two programs that ensures that the important properties of the so-called single-block encoding are also present in the large-block encoding. For this equivalence, LBE must not remove intermediate error states, a sequence of two transitions $error = 1$ and $error = 0$ for example must not be combined into the same block, as this would hide the intermediate error state. The result of LBE applied to the example program is shown in Fig. 1c.

2.6 Abstract Models

For model checking, enumerating over all concrete paths is often not feasible. A elegant way to get around this is to make use of a abstract representation of the program and then try to show that certain properties of the abstract system also hold for the concrete system.

In order to talk about such abstract representations, we introduce the concept of abstract models. An *abstract model* (A, ν, D) for a program $(T, C, C_0, \rightarrow)$ consists of

- a labeled transition system $A = (T, N, N_0, \rightarrowtail)$ over a set N of abstract nodes
- a the unwrap-function $\nu : N \rightarrow E$ that maps each abstract node to an abstract state out of a set E of abstract states

- an abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ that links the abstraction to the concrete representation of the program

This layer on top of the abstract domain might seem superfluous at first, but it will later allow us to describe splitting and slicing in a more rigorous way. The concretization function $[\![\cdot]\!] : E \to \mathcal{P}(C)$ assigns each abstract state the set of concrete states it represents. It is often convenient to directly concretize an abstract node, this can be done by sequentially composing the unwrap-function and the concretization function: $[\![\cdot]\!]_\nu := [\![\nu(\cdot)]\!]$. $\mathcal{E}$ is a lattice $(E, \sqsubseteq, \sqcup, \top, \bot)$ over the set E of abstract states. This allows us to make use of the lattice properties when reasoning about abstract models. Often we will choose a symbolic lattice where the elements are formulas. In this case $\top$ denotes a tautology and and $\bot$ denotes a contradiction.

The function $[\![\cdot]\!]$ can often also be written as a function $\sigma : E \to (X \to \mathbb{B})$ that maps each abstract state to a predicate over the program variables X. This predicate will be called *state formula*. For programs with control flow, the part of the state formula that corresponds to the data variables will be called *data-state formula*. We will restrict our considerations to concretization functions that can be expressed as state formulas.

To give a simple example for an abstract model, the control flow graph in Fig. 1b could be converted into an abstract model:

- $\rightarrowtail = \{(2, "error = 0", 3), (3, "i = 0", 4), \ldots, (9, "error = 1", 10)\}$
- $T = \{"error = 0", "i = 0", \ldots, "error = 1"\}$
- $N = \{2, 3, 4, 5, 7, 9, 11\}$
- $N_0 = 2$
- $\forall n \in N : \sigma(\nu(n)) \equiv (pc = n)$

In other words, the abstract states represent all concrete states that have a certain value for the program counter. A valid choice for the lattice then is $(\mathbb{N} \cap \{\top, \bot\}, \leq, max(\cdot, \cdot)), \top, \bot)$. This abstract model can already be useful, as locations that are not forward-reachable from the initial states can easily be identified.

2.6.1 Soundness

In order to infer properties of the concrete system from properties of an abstract model like in the example of the control flow graph, the abstract model needs to be sound. *Soundness* is achieved by satisfying the following conditions:

1. the initial abstract nodes of the abstract model are an over-approximation of the initial concrete states: $C_0 \subset \{[\![n_0]\!]_\nu \mid n_0 \in N_0\}$
2. the transfer function of the abstract model over-approximates the successors in the concrete system: $\left\{f_{\rightarrow}^{\tau}(n') \mid n' \in [\![n]\!]_\nu\right\} \subseteq \left\{[\![n']\!]_\nu \mid n' \in f_{\rightarrowtail}^{\tau}(n)\right\}$

A visualization of this the second relationship is shown in Fig. 2. First taking the (abstract) transfer function and then concretizing leads to a superset of the reverse order, i. e., first concretizing and then applying the (concrete) transfer function.

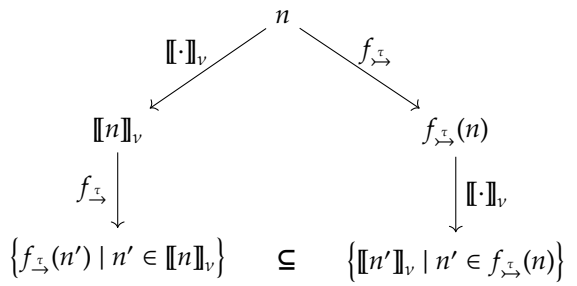


Figure 2: Visualization of the second soundness criterion

2.6.2 Abstract paths

An *abstract path* is a sequence of triples $(n_0, \tau_0, n_1), (n_1, \tau_1, n_2) \ldots, (n_{m-1}\tau_{m-1}, n_m)$ of abstract nodes and actions. The abstract path is *concretizable* if the path formula $(\sigma(\nu(n_0)) \wedge \Phi_0 \circ \sigma(\nu(n_1)) \wedge \Phi_1 \circ \cdots \circ \sigma(\nu(n_m)) \wedge \Phi_m)$ is satisfiable. The satisfiability of this path formula will ensure that there exists a concrete path $c_0, \tau_0, c_1, \tau_1, \ldots, \tau_{m-1}, c_m$ in the program. The inclusion of the state formulas is necessary to be sure that there is a concrete path that also represents the right abstract path, i. e., $c_i \in [\![n_i]\!]_\nu$ holds at every position $0 \leq i \leq m$.

An *abstract error path* $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{m-1}, n_m$ is an abstract path whose last state is an *abstract error node*, i. e., $error \wedge \sigma(\nu(n_m)$ is satisfiable. A sound abstract model proofs that a program fulfills a specification if it has no concretizable abstract error path. As a consequence, if the abstract model is sound and does not contain an abstract error node, there can be no abstract error paths and the program fulfills the specification.

Abstract error paths are also called *counterexamples*, as they have the potential of disproving that the program fulfills the specification. This does however only hold if the counterexample is not spurious. A counterexample is spurious if the corresponding path formula conjoined with the error predicate evaluated for the last state is unsatisfiable. This corresponds to the formula:

$$(\sigma(\nu(n_0)) \wedge \Phi_0 \circ \sigma(\nu(n_1)) \wedge \Phi_1 \circ \cdots \circ \sigma(\nu(n_m)) \wedge \Phi_m \wedge \sigma(\nu(n_{m+1}) \wedge error)$$

The check for the error predicate in the last abstract node may be omitted, provided that it is a node that has only concrete error states as concretization.

2.7 Constructing Abstract Models

For a given program and abstract domain there is a multitude of ways to build a sound abstract models. Often it is possible to state an algorithmic way to calculate the successors for an abstract state $e \in E$. This will lead to a transfer relation $\rightsquigarrow \in E \times T \times E$ that is sound if

$$\left\{f_{\rightarrow}^{\tau}(e') \mid e' \in [\![e]\!]\right\} \subseteq \left\{[\![e']\!] \mid e' \in f_{\rightsquigarrow}^{\tau}(e)\right\}$$

This transfer relation over abstract states that may correspond to an infinite system can be taken as a starting point to compute a finite abstract model that can then be used to proof certain properties of the underlying program.

Historically, different approaches have been developed to achieve this. In the area of static analysis, especially for use in compilers, fast lattice based approaches are used, which we will subsume under the name data-flow analysis. Another approach is traditionally used in model checking, where the (abstract) paths are explored in a tree-shape. Both of these techniques can be seen as edge cases of a unified approach, the configurable program analysis [1].

They start at the initial state and discover new abstract states from there in a stepwise fashion stepwise. Each abstract state that is discovered will be wrapped into a new abstract node. The partial abstract models that are built by such an algorithm in each step form an *abstract reachability tree* (ART, cf. [12]) or — more general — an *abstract reachability graph* (ARG).

2.7.1 Data-Flow Analysis

The lattice-based *data-flow analysis* generates the abstract model as a fixed point computation. Starting from the initial state, the successors are consecutively generated. Whenever a successor is encountered with the same location as an already explored state but with different abstract state, the wrapped abstract states are merged using the union operation that is given by the lattice structure of the abstract domain. The merged abstract state is wrapped into a abstract node that inherits its predecessors and successors from the two merged nodes. Note that due to the merge the tree-shape is lost, so we will have an ARG instead of an ART in most cases. The successors of a merged state are then explored again. This might result in consecutive merging of its successors with already existing states. This procedure is guaranteed to terminate if the lattice of the abstract states has finite height and if there are only finitely many locations in the control flow graph.

This is especially useful, e. g., for compiler optimizations, where arbitrary long waiting time for an analysis result is simply not acceptable. The restriction of the abstract domain to a lattice of finite (sufficiently small) height however limits the complexity of the properties that can be shown with that method. This is partly due to the fact that at the points where two states are merged, precision is lost. It is important that the merge of abstract states preserves soundness. For that one has to ensure that the concretization of the resulting abstract state is at least a superset of the concretizations of the abstract states that are joined:

$$[\![\text{merge}(n, n')]\!]_\nu \supseteq [\![n]\!]_\nu \cup [\![n']\!]_\nu$$

2.7.2 Model Checking

In model checking nodes with the same location are not merged. Thus the generated ARG remains in tree-shape, where each new abstract state is wrapped into a new abstract node. In order to get a finite abstract model, a fixed point is achieved by checking for each new abstract node n' whether the concrete states it represents are already completely entailed by one or several existing abstract nodes, which constitute the set N_c:

$$[\![n']\!]_\nu \subseteq \bigcup_{n \in N_c} [\![n]\!]_\nu$$

In this case, the new state is said to be covered by the other states. This is often indicated by a special coverage edge in the ARG. Coverage preserves soundness because the second soundness criterion holds if the edge into the covered node is redirected into edges into each of the covering nodes. Since the covering abstract nodes are not modified by this transformation, they need not be re-explored. This second way of looking at coverage can then be seen as a different kind of merging abstract nodes, but in order to preserve the advantage of having an ART instead of an ARG, the first way is normally used to express coverage.

As the union on the right hand side of the coverage condition is sometimes hard to compute, a stronger criterion can be used where the set N_c may only contain a single element. Since it might also not be feasible to calculate the subsumption over the concrete state sets, every preorder $\sqsubseteq$: $E \times E \to \mathbb{B}$ can be used instead, provided that it fulfills the following soundness criterion:

$$e_1 \sqsubseteq e_2 \Rightarrow [\![e_1]\!] \subseteq [\![e_2]\!]$$

The lattice can be chosen in a way that its subsumption relation is such a sound preorder.

2.7.3 Configurable Program Analysis

Both ways of reaching a fixed point can be expressed in a unified way by an algorithm where both the type of merge operation and coverage operation can be configured. This algorithm can then be set to work like data-flow analysis, like model checking or even offer a wide spectrum of new analysis between these two.

The algorithm takes as input the initial abstract state e_0 and a configurable program analysis (CPA). The program $L = (T, C, C_0, \to)$ that has to be checked is implicitly assumed to be given. A CPA is a tuple $(D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ that consist of an abstract domain D with a set E of abstract states, a transfer relation $\rightsquigarrow \in E \times T \times E$ and two operators $\mathsf{merge} : E \times E \to E$ and $\mathsf{stop} : E \times \mathcal{P}(E) \to \mathbb{B}$. The algorithm is consequently called the CPA algorithm. Note that since it was first published, several variants of the algorithm like CPA+ and the CPA++ have been formulated [13]. For now we will just consider the original version (cf. Algorithm 1).

The stop operator is used for coverage checks. The subsumption relation of the lattice of the abstract domain can be used to construct a stop operator and thus needs to respect the soundness equation for coverage from Section 2.7.2 unless soundness is not demanded.

The construction of the ARG is not explicit in the algorithm itself that just tracks the set of reached abstract states. It can however be built by wrapping the CPA into a special CPA $\mathbb{D}_{ARG}$ whose operators have the side-effect of modifying the transfer relation $\rightarrowtail$ of the ARG. The set of states for $\mathbb{D}_{ARG}$ can be expressed as $N = \mathbb{N} \times E$, so each ARG state is comprised of a node number and the wrapped abstract state. The transfer function of $\mathbb{D}_{ARG}$ is not a pure function, as it needs to keep track of node number in order to assign a new one to each new state. In this regard it becomes clear that $\mathbb{D}_{ARG}$ does not strictly adhere to the CPA concept, but is still useful for constructing the abstract model out of the performed steps in the CPA algorithm. Note the subtle distinction between an ARG and an abstract

Algorithm 1 CPA Algorithm, taken from [1]

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$, an initial abstract state $e_0 \in E$, where E denotes the set of elements of the lattice of D
Output: a set of reachable abstract states
Variables: a set reached $\subseteq E$, a set waitlist $\subseteq E$

```
1:  waitlist := {e_0}
2:  reached := {e_0}
3:  while waitlist ≠ ∅ do
4:    choose e from waitlist
5:    waitlist := waitlist \ {e}
6:    for all e' with e⇝e' do
7:      for all e'' ∈ reached do
8:        // combine with existing abstract state
9:        e_new := merge(e, e'')
10:       if e_new ≠ e'' then
11:         waitlist := (waitlist ∪ {e_new}) \ {e''}
12:         reached := (reached ∪ {e_new}) \ {e''}
13:     if ¬stop(e', reached) then
14:       waitlist := waitlist ∪ {e'}
15:       reached := reached ∪ {e'}
16:
17: return reached
```

model here. The ARG is a data structure that is used to represent the abstract model. In this sense we can use the terms interchangeably. For intermediate steps in the algorithm however, there might still be information missing in the ARG, so speaking of the abstract model here is somewhat misleading.

Another interesting aspect of the CPA framework is that it allows to combine different analysis for seperate abstract domains into one analysis. Given two CPAs $CPA_1 = (D_1, \rightsquigarrow_1, \mathsf{merge}_1, \mathsf{stop}_1)$ and $CPA_2 = (D_2, \rightsquigarrow_2, \mathsf{merge}_2, \mathsf{stop}_2)$ with abstract-state sets E_1 and E_2 and concrete-state sets C_1 and C_2, a combined $CPA_\times = (D_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times)$ over the domain $D_\times = (E_1 \times E_2, C_1 \times C_2, [\![\cdot]\!]_x : E_1 \times E_2 \to C_1 \times C_2)$ is easily obtained using the direct product. This can further be extended by the use of special operators that transfer information between CPAs and thus lead to an improvement of precision.

This composition is useful e. g. for tracking the program location as part of a separate LocationCPA. By doing so, it is possible to formulate analyses that do not depend on the program location and compose them with the LocationCPA in case this information is needed for the analysis.

2.8 Abstraction Refinement

While the algorithms presented in the previous section enable us to construct an abstract model, there is no guarantee that the result will be sufficient to decide whether the specification holds. If the abstract model satisfies the specification, so will the program. The reverse is in general not true, i. e., if the abstract model does not satisfy the specification, the program might still do. The reason for

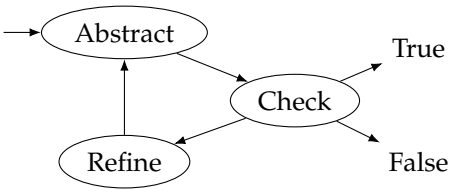


Figure 3: The Abstract-Check-Refine Paradigm. As long as the check of the constructed model is inconclusive, the model is refined and a new model is calculated

that is that the abstract model is only designed to be sound, but not necessarily precise.

2.8.1 CEGAR

In order to improve the abstract model, one needs a way to extract the information about why the analysis fails from the ARG. The analysis fails if there is a spurious counterexample, so the cause of infeasibility for an abstract error path can be used to make the abstract model more precise. This is the core idea behind *counterexample guided abstraction refinement* (CEGAR). Traditionally, this is done according to the abstract-check-refine paradigm [14, 15, 16], where the whole ARG is rebuild after the refinement (cf. Fig. 3).

This comes with an obvious drawback. A lot of computational results are discarded and need to be recomputed in every iteration. Techniques like lazy abstraction [17] try to mitigate this by refining the ARG on the fly without repeating the whole analysis. As a special case of this, the ARG can also be refined after the initial analysis is completed, a process that is commonly referred to as global refinement. In summary, CEGAR can mean one of the following refinement approaches:

1. change the inputs (e. g. abstract domain) and restart the analysis
2. improve the ARG during the analysis as soon as it is clear that the abstraction needs to be more precise
3. improve the abstraction after it is completed (global refinement)

2.8.2 Combination of CPA and CEGAR

CEGAR can be used together with the CPA algorithm, provided slight adjustments are made to the CPA algorithm. This modified version (cf. Algorithm 2) will be referred to as CPA* algorithm.

In order to be able to stop the analysis for the refinement and the continue it afterwards, the CPA* algorithm needs to return intermediate results. This can be achieved by providing a function abort that indicates whether the analysis shall be interrupted for a certain reached state. The CPA* algorithm will then return the set of reached states and the set of frontier states it has not yet considered. By providing these two sets as input, the algorithm can be continued after the refinement.

The algorithm for CEGAR (Algorithm 3) calls the CPA* algorithm in a loop and applies the refinement function in case a target state is present. If the refinement function fails to remove the target state from the ARG, the algorithm returns false, indicating that the program does not fulfill the specification. If the CPA algorithm finishes without any target state present, this means the program fulfills the specification and true will be returned. Note that the specification is given by the function isTargetState which can be seen as the algorithmic equivalent of the error predicate from Section 2.4.

The three ways to perform CEGAR from Section 2.8.1 are a subset of the possible configurations of the algorithms presented here.

The first way corresponds to a refine function that returns a reached set with a new, different starting state and an wait list containing this starting state. The abort function is set to always return false. Ideally, one wants to be able to change the other inputs to the CPA* algorithm, like the abstract domain itself or the transfer function. This is not possible in the CEGAR algorithm as presented here (Algorithm 3), but it can be easily extended in this direction.

For the second case, abort mimics the isTargetState function to ensure that the analysis is always stopped when a target state is reached. The refine function then tries to remove this target state.

The third approach is similar to the first, but the refine function is designed to remove all target states from the reached set and will not repopulate the wait list. This way, when not all target states can be removed, the CEGAR algorithm will return false after its next loop iteration.

Algorithm 2 CPA* Algorithm for use with CEGAR, adapted from [13]

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$, an initial abstract state $e_0 \in E$, where E denotes the set of elements of the lattice of D,
a list of abstract states waitlist,
a list of abstract states reached,
a function abort : $E \to \mathbb{B}$
Output: a set of reachable abstract states, a set of frontier abstract states
Variables: a set reached $\subseteq E$, a set waitlist $\subseteq E$

```
1:  while waitlist ≠ ∅ do
2:    choose e from waitlist
3:    waitlist := waitlist \ {e}
4:    for all e' with e⇝e' do
5:      for all e'' ∈ reached do
6:        // combine with existing abstract state
7:        e_new := merge(e, e'')
8:        if e_new ≠ e'' then
9:          waitlist := (waitlist ∪ {e_new}) \ {e''}
10:         reached := (reached ∪ {e_new}) \ {e''}
11:     if ¬stop(e', reached) then
12:       if abort(e') then
13:         return (reached, waitlist)
14:       waitlist := waitlist ∪ {e'}
15:       reached := reached ∪ {e'}
16:
17: return (reached, waitlist)
```

Algorithm 3 CEGAR Algorithm for CPA, adapted from [13]

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$,
an initial abstract state $e_0 \in E$,
a function $\mathsf{refine} : \mathcal{P}(E) \times \mathcal{P}(E) \to \mathcal{P}(E) \times \mathcal{P}(E)$,
a function $\mathsf{abort} : E \to \mathbb{B}$
a function $\mathsf{isTargetState} : E \to \mathbb{B}$
Output: **false** if abstract error state reachable, **true** otherwise
Variables: a set $\mathsf{reached} \subseteq E$, a set $\mathsf{waitlist} \subseteq E$,

```
1:  waitlist := {e_0}
2:  reached := {e_0}
3:  loop
4:    (reached, waitlist) := CPA*(𝔻, reached, waitlist, abort)
5:    if ∃e ∈ reached : isTargetState(e) then
6:      (reached, waitlist) := refine(reached, waitlist)
7:      if ∃e ∈ reached : isTargetState(e) then
8:        return false
9:    else
10:     return true
```

2.9 Interpolation

So far we have abstracted from how the refinement actually works. In general, the requirement for such a refinement is that, in case the discovered counterexample is spurious, some information has to be deduced automatically that can be used to improve the abstract model such that the counterexample is no longer present.

One way to get this information is through interpolation. Given two formulas A and B with $A \Rightarrow B$, the *Craig interpolant* [18] is a formula C that fulfills the following conditions:

- $A \Rightarrow C$
- $C \Rightarrow B$
- C does only contain atoms that occur both in A and B

In other words, the Craig interpolant C constitutes only the part of A that is necessary to conclude B.

In case of a spurious counterexample, the corresponding path formula (conjoined with the error predicate in the last node) is unsatisfiable.

Craig interpolation can be applied here by making use of the fact that the negation of an unsatisfiable formula is a tautology and vice versa. Thus a tautology $A \Rightarrow B$ holds iff $\neg(A \Rightarrow B)$ is unsatisfiable. This is equivalent to showing that $A \wedge (\neg B)$ is unsatisfiable. The Craig interpolant for an unsatisfiable formula $A \wedge B$ is a formula C such that:

- $A \wedge \neg C$ is unsat
- $C \wedge B$ is unsat
- C does only contain atoms that occur both in A and B

If the path formula is split into two parts, the Craig interpolant will contain the information that is necessary at the split position in the path in order to show that the whole path is unsatisfiable. This information can then be used for refinement. In the special case that B is already unsat, no knowledge from A has to be transferred, so the interpolant may be *true*. In the other special case where A is already unsat, the interpolant is allowed to be $false$.

Note that the existential quantifiers resulting from the sequential composition in the path formula can be removed by skolemization. This effectively reassembles static single assignment, where each intermediate state has its own set of fresh variable names. I.e., each transition formula Φ_i transfers from the set of variables X^i to the set X^{i+1}. We can bring the variables of the calculated interpolant back to X after interpolation with a simple substitution.

Sequences of interpolants can be generated by splitting the path formula between each pair of adjacent transitions. Given an unsatisfiable path formula $\Phi_0 \wedge \Phi_1 \wedge \cdots \wedge \Phi_m$ (in SSA form) for a path $(n_0, \tau_0, n_1)(n_1, \tau_1, n_2) \ldots (n_m, \tau_m, n_{m+1})$, the ith element of the sequence of interpolants $I_1, \ldots, I_m$ can be calculated as Craig interpolant of the formulas $\Phi_0 \wedge \cdots \wedge \Phi_{i-1}$ and $\Phi_i \wedge \cdots \wedge \Phi_m$.

As a special case, a sequence of interpolants is called inductive if $I_i \wedge \Phi_i \Rightarrow I_{I+1}$ holds for all $i \in \{1, \ldots, m\}$. Inductive interpolants can be calculated by replacing $\Phi_0 \wedge \cdots \wedge \Phi_{i-2}$ in the first formula by I_{i-1}. In other words, I_i can be calculated as Craig interpolant of $I_{i-1} \wedge \Phi_{i-1}$ and $\Phi_i \wedge \cdots \wedge \Phi_m$.

2.10 IMPACT

Impact[1] is a software verification approach for programs with control flow proposed by Kenneth McMillan in 2006 [19]. Being based on model checking, it generates the abstract model by unrolling the abstract state space in a tree-shape. Therefore coverage of nodes is used to reach a fixed point, as discussed in Section 2.7.2. Starting from an abstract node at the initial location, successors are explored according to the control flow. No further assumptions are made on the data variables, so the data-state formula is initially set to $\top$ for all abstract states. This way the soundness criterion is trivially fulfilled.

Besides expanding leave nodes and updating coverage, a refinement procedure can be used whenever an infeasible error path is present. The inductive sequence of interpolants is generated for the error path formula and the data-state formula for each node on the path is refined by conjunction with the corresponding interpolant. This is sound because the interpolants are proper invariants at their abstract nodes, a consequence of the tree-shape of the ARG. The data-state formula of the error node (and potentially other nodes as well) will become $\bot$, which means it can safely be removed from the ARG.

An illustration of different stages in the analysis for the example program from Fig. 1 is shown in Fig. 4. In Fig. 4a some nodes have been explored. The second node at location 4 is not expanded any further because it is covered by the first node at that location ($\top$ trivially implies $\top$). In Fig. 4b further expansion has discovered an error path. In Fig. 4c the refinement on the error path changed the data-state formulas according to the sequence of inductive interpolants. Note that this sequence is not unique and it depends on the interpolation method how this sequence will look like. E.g., both $i < 2$ and $i = 0$ are valid interpolants at

[1] Impact is actually the name of the tool, but we also use it synonymously for the approach

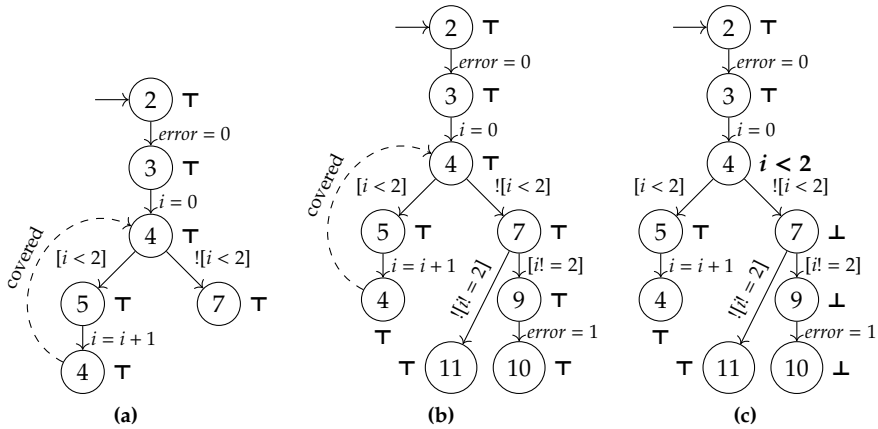


Figure 4: Different stages of checking the example program from Fig. 1 with Impact. Bold formulas next to the nodes represent the data-state formula, the location is written inside the nodes. Coverage is drawn with dashed edges.

location 4 in the error path. Coverage at location 4 has been removed in Fig. 4c, because $\top \Rightarrow i < 2$ does not hold. Amongst others, the error state is infeasible (data-state formula $\bot$) and can be removed, indicating a successful refinement. Impact would then continue with expanding the leave node at location 4. For this example, it would end with a sound abstraction that proofs the program correct.

The Impact algorithm can be expressed in the CPA framework [20].

2.11 Adjustable-Block Encoding

Impact can be used with large-block encoding, as this will only affect the program representation that is treated as input to the algorithm. This will result in fewer nodes and especially less interpolation steps for each refinement, which can increase performance [20]. The main limitation for the block size of LBE are the loop heads, as unbounded loops cannot be expressed in closed form by formulas of first-order logic. Impact however does model checking with a tree-shaped ARG, so loops are naturally unrolled in the abstract model. By moving the block encoding into the abstract model, different block sizes than LBE are possible, which has the potential of being more efficient. This approach called *adjustable-block encoding*(ABE) was initially developed for predicate abstraction [10], but can also be applied to Impact [20].

In ABE, states are distinguished into two categories, abstraction states and non-abstraction states. This can be modelled by an operator, the so-called block operator $blk : E \times T \rightarrow \mathbb{B}$, which returns true if the successor state under a certain transition is an abstraction state. The interpolants along a path are only calculated for the nodes with abstraction states.

The state formula of the intermediate non-abstraction states is given implicitly, since it is not of importance for the analysis. It is enough to know that for the intermediate states the state formula could be calculated such that the whole abstract model is sound. For example, in predicate abstraction with ABE the non-abstraction state saves the state formula of the previous abstraction state

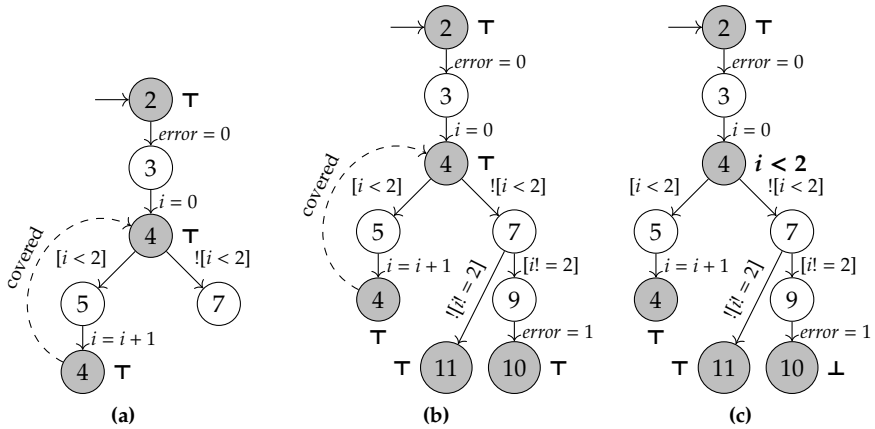


Figure 5: The IMPACT analysis from figure 4 but with adjustable-block encoding enabled. The block operator is chosen to mimic large-lock encoding. Gray nodes contain abstraction states, the others contain non-abstraction states.

and the path formula to itself from there. Both combined could be used to determine the exact state formula of this non-abstraction state, but this is never explicitly done. Instead this information is carried on to the next abstraction state, where it is used to calculate the computationally expensive predicate abstraction.

The merge operator has to be adapted for ABE in a way that it merges two nodes if their abstract states are non-abstraction states. This leads to ARGs where the tree-shape does only hold if just the abstraction states are considered. The general shape will be an acyclic graph. For calculating the path formulas at points where two non-abstraction states transition into the same non-abstraction state a disjunction can be used. This then reassembles the path formula structure of LBE, where a single path formula can express multiple branches at once.

An example of IMPACT with ABE configured to emulate LBE is shown in Fig. 5. The gray abstraction nodes are the same nodes we would get with LBE.

CPA Adjustable-block encoding can be realized using the CPA $\mathbb{D}_{ABE} = (D_{ABE}, \rightsquigarrow_{ABE}, \mathsf{merge}_{ABE}, \mathsf{stop}_{ABE})$ that has been developed for predicate abstraction with ABE [10]. It is defined using a different version of the CPA algorithm where a set Π of predicates called the precision is tracked for each location. This precision is used by the analysis to perform predicate abstraction in the transfer relation. Note that we will be using formulas here instead of sets of predicates as in the original formulation. This is equivalent as the conjunction of all predicates in a set of predicates yields an equivalent formula.

The abstract domain consists of abstract states $(l, \psi, l^{\psi}, \varphi)$, where l^{ψ} is the location of the last abstraction state and φ is a path formula that represents the path from the last abstraction state to the current state. l is the current location and ψ a state formula. In predicate abstraction, for every abstraction state ψ will result from the predicate abstraction calculation.

The transfer relation contains the transition $e_1 \overset{\tau}{\rightsquigarrow} e_2$ for states $e_1 = (l_1, \psi_1, l_2^{\psi}, \varphi_1)$ and $e_2 = (l_2, \psi_2, l_2^{\psi}, \varphi_2)$ exactly when:

$$\begin{cases} l_1 \xrightarrow{\tau} l_2 \wedge (\varphi_2 = \top) \wedge (l_2^{\psi} = l_2) \wedge \psi_2 = (\psi_1 \wedge \varphi_1 \circ \varphi_\tau)^{\Pi(l_2)} & \text{if } \mathsf{blk}(e_1, \tau) \\ l_1 \xrightarrow{\tau} l_2 \wedge (\varphi_2 = \phi_1 \circ \phi_\tau) \wedge \psi_2 = \psi_1 \wedge l_2^{\psi} = l_1^{\psi} & \text{otherwise} \end{cases}$$

The result of the predicate abstraction $(\cdot)^{\Pi(l_2)}$ in the transfer relation will always be $\top$ or $\bot$ in case that the precision $\Pi(l_2)$ is an empty set. φ_τ is the data transition formula that describes the transition τ. When the blk operator returns true, e_2 is an abstraction state. Note that $(l_2^{\psi} = l_2) \wedge (\varphi_2 = \top)$ is strictly not sufficient to distinguish abstraction states from non-abstraction states, so this property has to be tracked otherwise, e. g. by $\mathbb{D}_{ARG}$.

The merge operator merge_{ABE} for states e_1 and e_2 as above is defined as:

$$\mathsf{merge}(e_1, e_2) = \begin{cases} (l_2, \psi_2, l_2^{\psi}, \varphi_1 \vee \varphi_2) & \text{if } (l_1 = l_2) \wedge (\psi_1 = \psi_2) \wedge (l_1^{\psi} = l_2^{\psi}) \\ (l_2, \psi_2, l_2^{\psi}, \varphi_2) & \text{otherwise} \end{cases}$$

The stop operator stop_{ABE} is defined as follows:

$$\mathsf{stop}((l, \psi, l^{\psi}, \varphi), \mathsf{reached}) = \begin{cases} \mathit{true} & \text{if } \exists (l', \psi') \in \mathsf{reached}: \\ & l \in \{l', l_\top\} \wedge (\psi \circ \varphi \Rightarrow \psi' \circ \varphi) \\ \mathit{false} & \text{otherwise} \end{cases}$$

The blk operator in the transfer relation can be set to blk_{sbe}, which always return true. As a result there are only abstraction states, so l^{ψ} is always equal to l and φ is always $\top$. In this case, the CPA $\mathbb{D}_{ABE}$ is equivalent to $\mathbb{D}_K$, provided that the initial abstract state has the data-state formula $\top$.

Another blk operator is blk_{lbe}, which returns true only for initial states, error states and loop heads.

2.12 Slicing Abstractions

Impact makes use of a tree-shaped ARG for its refinement and is thus not suited in cases where the ARG does not form a tree. A more general refinement approach that works for all graph-like ARGs is slicing abstractions [4]. Here one makes use of the fact that it is always sound to split a node n in the ARG into two nodes n' and n'' such that the concretizations of the splitted nodes are a superset of the concretization of the original node:

$$[\![n]\!]_\nu \subseteq [\![n']\!]_\nu \cup [\![n'']\!]_\nu$$

For the special case where the set equality holds in this equation, the set of concrete paths represented by the abstraction does not change. This is always the case when the splitting is performed in a way that can be expressed by a predicate $\psi(X)$. Suppose the node n has the state formula $\sigma_n(X)$, then splitting with the predicate ψ will lead to two states with state formulas $\sigma_n \wedge \psi$ and $\sigma_n \wedge \neg\psi$ respectively. Together these states will always represent the same concrete states as the original state:

$$[\![\sigma_n \wedge \psi]\!] \cup [\![\sigma_n \wedge \neg\psi]\!] \equiv [\![\sigma_n \wedge \psi \vee \sigma_n \wedge \neg\psi]\!] \equiv [\![\sigma_n \wedge (\psi \vee \neg\psi)]\!] \equiv [\![\sigma_n]\!]$$

When a node is split, transitions have to be copied from the original node. For self loops, all possibilities have to be considered, e. g. if a node has the self loop $n \overset{\tau}{\rightarrowtail} n$, the transition relation after splitting n into n' and n'' has to contain $n' \overset{\tau}{\rightarrowtail} n'$, $n'' \overset{\tau}{\rightarrowtail} n''$, $n' \overset{\tau}{\rightarrowtail} n''$ and $n'' \overset{\tau}{\rightarrowtail} n'$. This is depicted in Fig. 6.

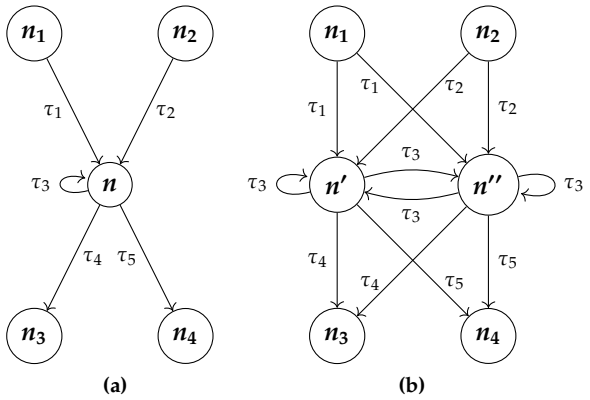


Figure 6: Example of splitting a node n into two nodes n' and n'' in an abstract model. **(a)** is before the split, **(b)** afterwards.

Splitting alone is not enough for an effective refinement, as it does not reduce the number of abstract paths. After splitting however, some of the edges in the transition relation might actually become infeasible and every path that contains them automatically becomes infeasible as well. These edges can therefore be removed from the ARG. As further consequence, some nodes might even become unreachable from the initial node in the ARG and can also be removed. It is sound to remove an edge $n \overset{\tau}{\rightarrowtail} n'$ if the associated path formula $\sigma(\nu(n)) \wedge \Phi_\tau \circ \sigma(\nu(n'))$ is infeasible.

2.12.1 For programs: Kojak

Abstraction slicing can be used as a refinement for programs, which contain information about the control flow. An example is the Kojak algorithm [6], that is also implemented in the software model checker Ultimate [7]. Kojak starts with an initial abstract model that reassembles the CFA, where each abstract state has the state formula $\top$. As long as an infeasible error path is present, a sequence of inductive interpolants is generated. For interpolation, it is sufficient to use the data transition formulas in the path formula, e. g.:

$$(\sigma(\nu(n_0)) \wedge \varphi_0 \circ \sigma(\nu(n_1)) \wedge \varphi_1 \circ \cdots \circ \sigma(\nu(n_m)) \wedge \varphi_m) \tag{1}$$

The reason for this is that the control flow integrity is already assured by the initial abstract model. Each node on the path is then split according to the corresponding interpolant. After splitting, the edges are checked for feasibility in a slicing step. It is sufficient to use the data transition formulas in the slicing path formula, e. g., $\sigma(\nu(n)) \wedge \varphi_\tau \circ \sigma(\nu(n'))$, by the same argument that holds for the interpolation. The inductive property of the interpolants can be used to

show that each refinement ensures a certain progress, i. e., the same error path will not be discovered again [6].

The algorithm continues splitting and slicing until either a concretizable error path is discovered – in which case the program does not fulfill the specification – or no more error states are present (reachable from the initial node) in which case the program satisfies the specification.

Note that original formulation of Kojak uses LBE as an optimization. This leads to a smaller abstract model, where there are fewer interpolants per error path. Another effect is that the path formulas describe more than one transition, so they are generally harder to proof infeasible by a solver. But this also allows offloading work from the interpolation to the slicing step. For example, imagine a sequence of two transitions $\tau_1 = \text{"}[x == 0]\text{"}$ and $\tau_2 = \text{"}[x == 1]\text{"}$ with $n_0 \overset{\tau_1}{\rightarrowtail} n_1$ and $n_1 \overset{\tau_1}{\rightarrowtail} n_2$. The formula for the sequential composition of both transitions is infeasible, independent on the state formulas, whereas the formulas for the individual transitions could be feasible. In the case where the transitions are considered separately, an interpolant like $x = 0$ has to be added before the slicing can be successful. When both transitions are treated as a single transition, this interpolation step is not necessary.

The application of the Kojak algorithm to the example program in Fig. 1 is shown in Fig. 7. In Fig. 7a the initial abstract model is shown, which is just the CFA of the example program from Fig. 1b annotated with a state data formula $\top$ for every abstraction node. An error path through nodes 2,3,4,7,9,10 is present. As this path is infeasible, interpolants are calculated. For nodes 2 and 3 these are $\top$. At node 4 a valid interpolant is $i = 0$. For locations 7,9,10 the interpolant is $\bot$ since the transition from location 4 to location 7 is incompatible with the interpolant at location 4. Figure 7b shows the state of the abstract model after splitting according to these interpolants. Only node 4 is split into nodes 4′ and 4′′. The other nodes on the error path are not split, because doing so with an interpolant that is $\top$ or $\bot$ has no effect (other than changes of node labels) after the slicing step. The effect of the slicing step can be seen in Fig. 7c. Edges from nodes 4′ to 7 and 3 to 4′′ are removed because they are not compatible with the interpolants added to the split nodes. The effect of this refinement is that the original error path is no longer present. The shortest error path is now longer than the previous one, as can be seen in Fig. 7d where the nodes where merely repositioned. The refinement effectively unrolled one loop iteration, so at least two more refinements would be necessary to proof the program correct.

2.12.2 For transition systems: SLAB

Kojak makes use of the control flow in order to find a suitable initial abstract model. This however limits the kind of transition systems that can be analyzed to programs, which contain control-flow information.

The first approach that introduced the idea of abstraction slicing for software model checking does not depend on control flow [4] and is implemented in the tool SLAB [5]. It is intended for checking infinite-state concurrent systems and contains several optimizations, which will not be covered in full extend here. We will limit the description to the basic concept and only those optimizations that are important for further understanding. The procedure will nonetheless be referred to by the name SLAB.

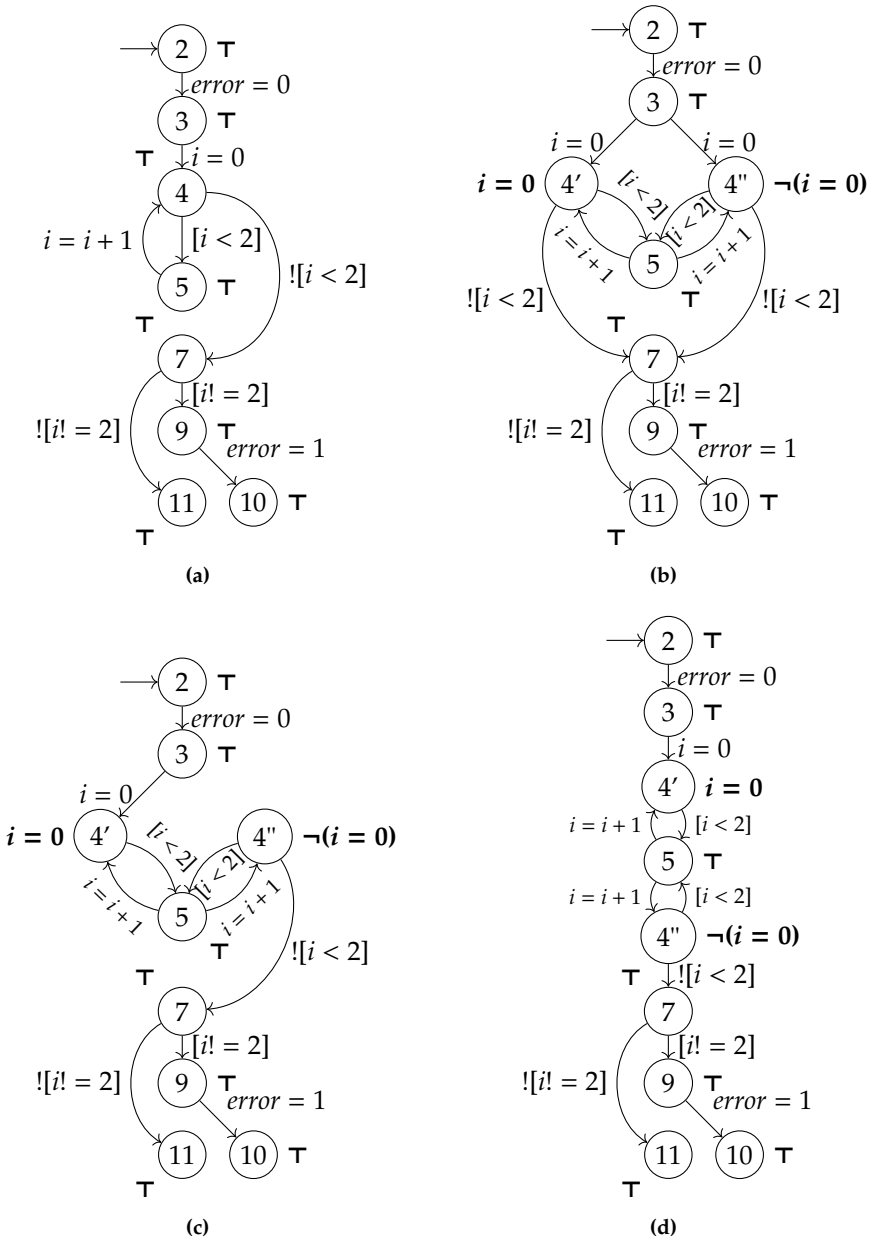


Figure 7: First refinement of Kojak applied to the example program from Fig. 1. **(a)** shows the initial abstract model, **(a)** is after the first split. In **(c)** infeasible edges have been sliced. **(c)** is a rearrangement of the nodes to show that the minimal error path length changed.

For the initial abstract model, SLAB uses abstract states with state formulas that consist of conjunctions of two predicates and their negations. One of them is the predicate *init*(*X*) that identifies the concrete states that are considered

as initial states. The other one is the predicate $error(X)$ that defines which concrete states are considered error states. In total there are four different ways to make a conjunction out of these two predicates and their negation. The initial abstract model contains one node for each of these four abstract states. The transition relation starts with all possible transitions between these four nodes: $\rightarrowtail = N \times T \times N$. The two nodes where the predicate $init(X)$ appears in unnegated form constitute the set of initial nodes.

Regarding soundness, every concrete state has to be represented by exactly one of these four nodes, as the disjunction of their state formulas is equivalent to $\top$. A transition from a node to another node is always possible for an arbitrary τ. This means that for every concrete path there is an abstract path, which means this initial abstract model is indeed sound.

The initial abstract model in its general form can be seen in Fig. 8a. The edges are labeled with sets of actions now, as due to the lack of control flow the number of edges between two nodes is not limited. The state formulas are written next to their node. In most cases the formula $init \wedge error$ is unsatisfiable. Otherwise there would be an initial concrete state that already violates the specification. The initial abstract model can be simplified by removing this node, resulting in the abstract model shown in Fig. 8b. It can further be simplified by slicing those transitions that are infeasible. The exact result of this depends on the program, an exemplary slicing result as it occurs for most programs with control flow is shown in Fig. 8c. Here the middle node n_2 represents all intermediate program states and therefore also aggregates most of the transitions from the transition set T in a self loop. By splitting, the set of transitions for this self loop usually gets smaller, an more detailed abstract model is generated. One possibility of such a split is shown in Fig. 8d.

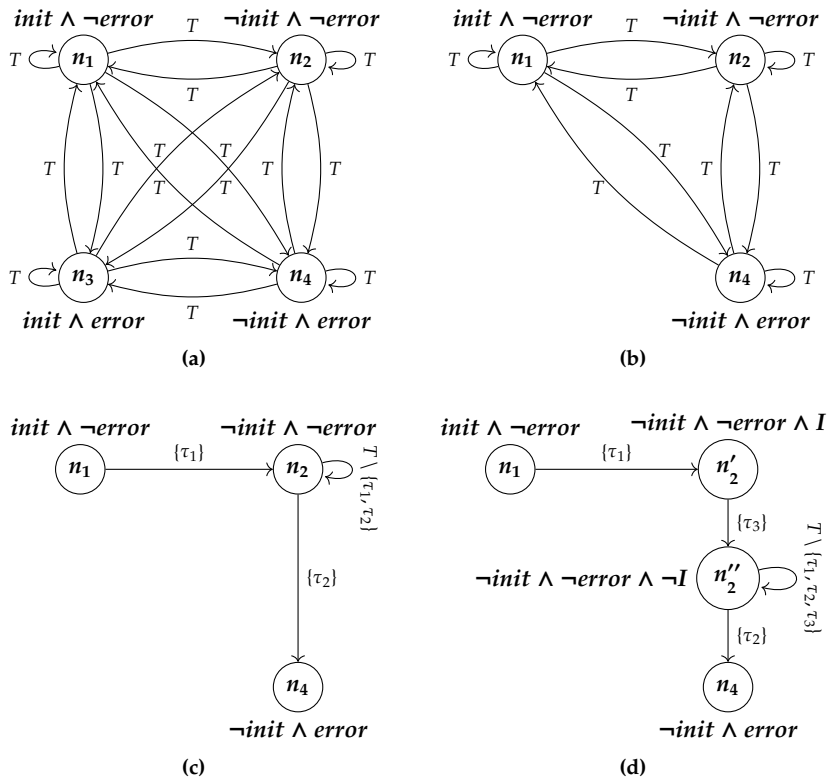


Figure 8: Visualization of different phases in SLAB. **(a)** shows the unsimplified initial abstract model, in **(b)** node n_3 is removed since it is normally infeasible. **(c)** shows an exemplary result of slicing edges in **(b)**. A possible result of splitting and slicing is shown in **(d)**

SLAB in its original form does not use inductive interpolants for splitting. Instead, the minimal spurious subpath in the error path is identified. This subpath is then split before its last transition and a single Craig interpolant is calculated for these two formulas. The result is a single interpolant at the second last state in the subpath. The intention behind this is to find a split that effects many abstract paths at once, not just the error path currently in consideration.

Abstraction slicing works for a wide range of splitting criteria, so the exact interpolation method can be seen as a configurable component. We will deviate from the original description and also use inductive interpolants for SLAB, as this will make it easier to compare it to the other techniques like Kojak and Impact that also use this kind of interpolation.

Up to now, only the basic form of SLAB has been described. Among the different optimizations, there is one that can reduce the node count of the abstract model by bypassing intermediate nodes, much like Kojak uses LBE for the same purpose.

Whenever there is a node n that has incoming transitions T_i from only a fixed node n_i and outgoing transitions T_o to only a fixed node n_o, n can be bypassed. This happens by adding so-called bypass transitions to the set of transitions:

$$T' = T \cup \{\tau_i \circ \tau_o \mid \tau_i \in T_i, \tau_o \in T_o\}$$

For each possible pair $(\tau_1, \tau_2) \in T_i \times T_o$, the transition $n_i \overset{\tau_i \circ \tau_o}{\rightarrowtail} n_o$ is added to the transition relation, while the transitions $n_i \overset{\tau_i}{\rightarrowtail} n$ and $n \overset{\tau_o}{\rightarrowtail} n_o$ are removed. Since the node n is then unconnected to the rest of the abstract model, it can also be removed from the set of nodes. The semantics of the bypass transitions is naturally defined by the sequential composition of the corresponding transition predicates:

$$\Phi_{\tau_i \circ \tau_o} = \Phi_{\tau_i} \circ \Phi_{\tau_o}$$

For an example of bypassing, look at node n_2' in Fig. 8d. This node can be removed if the bypass transition $n_1 \overset{\tau_1 \circ \tau_3}{\rightarrowtail} n_2''$ is added.

The application of SLAB to the example program in Fig. 1 will look essentially like shown in figure 7, where $\tau_1 = \text{"}error = 1\text{"}$ is the first action from the initial node and $\tau_2 = \text{"}error = 1\text{"}$ is the action that leads to the error location. The transition predicate for τ_1 then is $pc = 2 \wedge pc' = 3 \wedge error' = 0$ and for τ_2 it is $pc = 9 \wedge pc' = 10 \wedge error' = 1$. The error path $(n_1, \tau_1, n_2), (n_2, \tau_2, n_4)$ in Fig. 8c is spurious because the program counter values of the two transitions are not compatible. As a consequence, the interpolant for splitting will be $pc = 3$. This way, consecutive splittings will uncover those parts of the control flow that are necessary to verify the program.

3 Slicing Abstractions as CPA

So far, two different existing approaches for abstraction slicing have been presented. One that makes use of the control flow of a program and one that does not. The CPA framework (in combination with the CEGAR algorithm) can be used to express various analyses using the same algorithmic setup. This makes it possible to find differences, common aspects and also aids a better comparison by varying joint building blocks [13, 21].

In order to be able to compare these two abstraction slicing techniques to each other as well as to other state of the art approaches, they can be fitted into the CPA framework. This consists of two parts. The first is to determine the inputs for the CPA* algorithm that will lead to the desired abstraction. The second part is to determine the inputs for the CEGAR algorithm, especially the refinement procedure that will perform the splitting and slicing steps. Ideally this refinement is designed in a way that it is the same for both CPAs. This might even make it possible to use the refinement as a building block and combine it with already existing CPAs to from new analyses.

Both analyses will be configured as a global refinement. The CPA algorithm builds the initial (and sound) abstract model and then the global refinement will attempt to remove all error states or find a feasible counterexample.

3.1 Kojak Analysis

The Kojak analysis will first be formulated in a form that is equivalent to single-block encoding. After that we add adjustable-block-encoding in order to be able to comparable our implementation to the original Kojak implementation which uses large-block encoding.

3.1.1 With Single-Block Encoding

CPA The Kojak analysis has the CPA $\mathbb{D}_K = (D_K, \rightsquigarrow_K, \mathsf{merge}_K, \mathsf{stop}_K)$, where the abstract domain is $D_K = (C, [\![\cdot]\!]_K, (E_K, \sqsubseteq_K, \sqcup_K, \top_K, \bot_K))$. The abstract domain for the analysis has to keep track of the program counter as well as the data-state formula. Therefore an abstract state in E_K can be modeled as a tuple (l, ψ). l is an integer (or the special element $l_\top$) indicating the value of the program counter and ψ is a formula over the data-state variables V. $l_\top$ is needed to be able to define a proper lattice. Two data-state formulas who are equivalent are considered to be the same element. This removes syntactic differences in formulas, e. g. $a \wedge \neg a$ and $\top$ are equivalent and therefore $(0, a \wedge \neg a)$ is considered to be the same abstract state as $(0, \top)$. The concretization function $[\![\cdot]\!]$ for this set of abstract states is given by the following identity:that

$$c \in [\![(l, \psi)]\!] \quad \Leftrightarrow \quad c(pc) = l \wedge \psi[c(X)/X]$$

The reached set and the wait list for the CPA* algorithm are set to $\{(l_0, \top)\}$. In order to configure a global refinement, the abort function always returns false. The transfer relation does not need to change the data-state formula, it just has to ensure that the control flow is respected:

$$(l, \psi) \overset{\tau}{\rightsquigarrow} (l', \psi') \quad \Leftrightarrow \quad l \xrightarrow{\tau} l' \wedge (\psi \equiv \top) \wedge (\psi' \equiv \top)$$

The merge operator always merges nodes provided they share the same location:

$$\mathsf{merge}((l, \psi), (l', \psi')) = \begin{cases} (l, \psi \vee \psi') & \text{if } l = l' \\ (l', \psi') & \text{otherwise} \end{cases}$$

The stop operator marks a state as covered if there is a state in the reached set that has the same location and whose data-state formula is implied by the data-state formula of the covered state:

$$\mathsf{stop}((l, \psi), \mathsf{reached}) = \begin{cases} \mathit{true} & \text{if } \exists (l', \psi') \in \mathsf{reached} : l \in \{l', l_\top\} \wedge (\psi \Rightarrow \psi') \\ \mathit{false} & \text{otherwise} \end{cases}$$

When the CPA* algorithm is executed with this inputs, the merge operator does not return a different abstract state when given two states with the same location. This is because the data-state formula does not actually change. The coverage relation induced by the stop operator marks all states with same location. We can use this to construct the desired abstract model that reassembles the control flow graph if we remove the covered nodes and deflect their incoming edges to the node which covers them as described in Section 2.7.2.

We consider this post-processing step to be a implementation detail of the CPA $\mathbb{D}_{ARG}$ into which the CPA $\mathbb{D}_K$ is wrapped in the analysis. The merge

operator of this $\mathbb{D}_{ARG}$ can alternatively complement the merging behavior of merge_K in order to truly merge states with the same location.

When the CPA* algorithm given these inputs finishes, the ARG that is constructed by $\mathbb{D}_{ARG}$ will reassemble the initial abstract model of Kojak.

Refinement The function isTargetState that is given as input to the CEGAR algorithm (Algorithm 3) will return true for a given abstract state $E = (l, \psi)$ if the data-state formula and the error predicate are compatible, e. g. if $\psi \wedge \mathit{error}$ is satisfiable. If the abstract states are guaranteed to be either pure error states or do not contain any concrete error state, then this is equivalent to demanding that $\psi \Rightarrow \mathit{error}$ holds.

The refinement function refine_{SliAbs} (cf. Algorithm 4) performs splitting and slicing until a feasible counterexample is found or no more target states are present in the calculated reached set. This reached set is constantly updated to account for the changes that are due to the splitting and slicing procedures. For this update, a reachability analysis has to be performed on the ARG, starting from the initial node.

refine_{SliAbs} contains several procedures. getErrorPath returns a list of abstract states that describe the error path. sat is a decision procedure that decides whether such a path is feasible. In case the error path is infeasible, the procedure getInterpolants calculates the sequence of inductive interpolants. The procedure split uses this sequence to split the nodes on the path while copying the edges accordingly (cf. Fig. 6). After this, the procedure sliceEdges removes all edges that are infeasible from the ARG.

Note that, as slicing can remove edges between two parts of the ARG that might or might not be connected somewhere else, reachability cannot be decided locally. The time complexity of this calculation is linear in the edge count of the ARG and has no time-intensive operations like solver calls, thus it is unlikely to be a bottleneck for the performance of the analysis.

Solver calls occur for interpolation and when checking each edge in the slicing. Since only some of the edges are actually affected by the added interpolation predicates, the slicing can be optimized to only call the solver in cases where the result is unknown.

3.1.2 With Adjustable-Block Encoding

The CPA $\mathbb{D}_K$ looks very similar to $\mathbb{D}_{ABE}$. In fact, they are so similar that $\mathbb{D}_{ABE}$ can be seen as a generalization of $\mathbb{D}_K$. If we choose an blk_{sbe} and an empty precision for $\mathbb{D}_{ABE}$, all states will be abstraction states and the state formulas ψ will remain $\top$ as desired.

The reuse of $\mathbb{D}_{ABE}$ has the advantage that we can use different block operators. When using blk_{abe}, we get adjustable-block encoding for free. The result of running the analysis with this block operator is shown in Fig. 9a. The abstraction states are labeled with their data-state formula, which is always $\top$. The other components of the abstract state are not shown, as they are not of interest for Kojak.

Refinement In order to make refine_{SliAbs} work on abstract models that contain non-abstraction states, some of the procedures in the refinement have

Algorithm 4 refinement refine_{SliAbs} for abstraction slicing

Input: a list of abstract states waitlist,
a list of abstract states reached
Output: **false** if abstract error state reachable, **true** otherwise
Variables: a map interpolants $\subseteq E \to \mathcal{F}(X)$ of abstract states to formulas, a list errorPath of abstract states, a list of abstract states currentReached

```
1:  currentReached = reached
2:  while ∃e ∈ currentReached : isTargetState(e) do
3:    errorPath = getErrorPath(reached)
4:    if sat(errorPath) then
5:      return (waitlist, currentReached)
6:    interpolants = getInterpolants(errorPath)
7:    for all (n, I) ∈ interpolants do
8:      split(n, I)
9:    currentReached = updateReachedSet(currentReached)
10:   for all n ∈ currentReached do
11:     sliceEdges(n)
12:   currentReached = updateReachedSet(currentReached)
13: return (waitlist, currentReached)
```

to be adapted. For one, the procedure getErrorPath is modified such that it only returns the abstraction states on the error path. The interpolation step in getInterpolants then needs to construct the path formulas between these abstraction states and use them to calculate the inductive interpolants at the abstraction states in the path.

Bigger changes are necessary for the splitting and slicing steps, since the concept of what an edge for slicing is has changed. As the abstraction nodes are what is important for the analysis, it is of interest whether two abstraction nodes are connected via non-abstraction nodes. We call the non-abstraction nodes that lie between these two nodes *segment* and say that there is a *segment edge* between the two states. Each segment has an abstraction node at which it starts and an abstraction node at which it ends. For example, in 9a there is a segment edge between the node at location 4 and the node at location 10 with a segment containing the nodes at locations 7 and 9. So instead of slicing the usual edges between abstract nodes (which could be non-abstraction or abstraction nodes), the whole segment edges have to be checked for feasibility in the slicing step. Eventually, the abstraction has to be changed in a way that the segment edge is removed. The latter has to be done in a way that does not change the other segment edges, which is not always trivial. We can see this if we look at a butterfly-like graph as shown in Fig. 9b. Suppose the slicing determined that the segment edge between n_1 and n_3 is infeasible. In order to remove the segment edge, it is clear that the direct edge from n_1 to n_3 needs to be removed. Also, the segment has to be disconnected from the abstraction nodes somehow. But neither the edge from n_1 to n_a nor the edge from n_b to n_3 can be removed, as this would also affect the segment edges between n_1 and n_4 or n_2 and n_3.

One way of solving this is to copy the segment immediately before slicing and thus eliminate the butterfly configuration (*copy-on-slice*). An example for this is shown in figure 9c. Here the segment is duplicated such that each

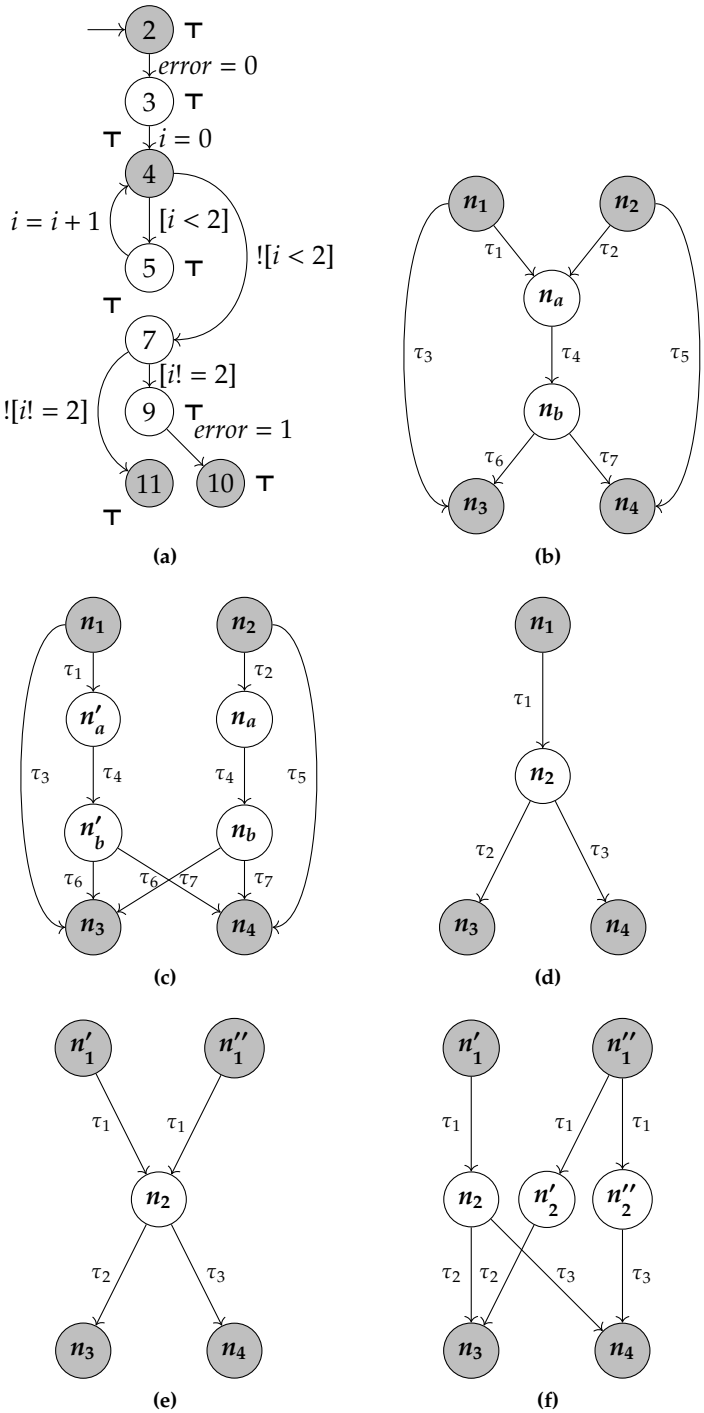


Figure 9: Various examples for explaining the design of Kojak with ABE

non-abstraction node is only in segments that start at the same abstraction node. If this property holds for every non-abstraction state, slicing a segment edge can be performed by cutting the right regular edges. In the example of Fig. 9c, these are the edges from n_1 to n_3 and from n'_b to n_3.

The other way is to avoid having non-abstraction states that are in segments with different abstraction states at the beginning. Due to the way how $\mathbb{D}_{ABE}$ is designed, the initial abstract model is already in this form. Violations then occur when the splitting behavior is not changed compared to the SBE case. A solution here is to copy the segment edges instead of the regular edges when splitting (*copy-on-split*).

In Fig. 9d there is an example of how a part of the abstract model looks like after the CPA* analysis finishes. Suppose the abstract node n_1 is split into n'_1 and n''_1 like in the SBE case, then the result would look like Fig. 9e, where slicing the segment edge from n'_1 to n_3 is a nontrivial task. Instead we can split n_1 like shown in Fig. 9f, copying every segment that starts at n_1 and ends at n_3 over to n''_1 while replacing n_1 by n'_1. This way slicing the segment edge from n'_1 to n_3 is straightforward. A drawback is the multiple copies of non-abstraction states that get generated this way.

3.1.3 Interprocedural Analysis

So far, we have not discussed the call of procedures. Since the call of a procedure can happen at multiple locations, there are also multiple possible return positions after the procedure call. This is normally indicated by a special return edge in the CFG. This allows the CFG to be more compact, since inlining the procedure leads to multiple copies of the body. Inlining also has the disadvantage that recursive calls cannot be handled this way.

For the case where the procedure calls are not inlined in the CFG, we need to extend the CPA for Kojak in order to remain precise. Otherwise a path where we exit the procedure at the wrong return node will not be recognized as infeasible. This can be fixed by taking advantage of the CPA framework. A call-stack CPA D_{CS} that keeps track of the call stack can be added to D_{ABE} using CPA composition. This way, when the CPA* algorithm constructs the initial abstract model, states inside the body of the procedure that belong to different procedure calls have a different call-stack state and are therefore not merged. In the resulting initial abstract model, the procedure will now be inlined and soundness is restored.

The CPA* algorithm will not terminate with the new composite CPA as input if we want to analyze a recursive program. One way of fixing this is representing the call stack symbolically as part of the transition formulas instead of using the call-stack CPA D_{CS}. As this would reduce the interprocedural analysis to an intraprocedural analysis and thereby also occlude important information about the procedural nature of the program structure, we are not going to explore this any further. After all, such a symbolic call-stack representation is orthogonal to Kojak and might be useful also for other analysis.

Note that the software model checker Ultimate uses a special *recursive CFG* and nested word automata with nested (tree) interpolants for proving recursive programs [7, 22, 23].

3.2 SLAB Analysis

For the analysis reassembling SLAB we will procede in a similar manner as with Kojak. First, we define an analysis that has edges that only contain a single transition. After that, we add an optimization that is comparable to large-block encoding.

3.2.1 With Single-Block Encoding

CPA The CPA $\mathbb{D}_S = (D_S, \rightsquigarrow_S, \mathsf{merge}_S, \mathsf{stop}_S)$ for SLAB does not need to track the program counter explicitly. Therefore the abstract states can be chosen as state formulas ψ, which can contain information about the program counter symbolically. The subsumption relation $\sqsubseteq_S$ for the lattice $\mathcal{E}_S = (E_S, \sqsubseteq_S, \sqcup_S, \top_S, \bot_S)$ of the abstract domain $D_S = (\mathcal{E}_S, [\![\cdot]\!]_S, C)$ generates a flat lattice where — apart from $\top_S$ and $\bot_S$ — only equisatisfiable formulas are subsumed:

$$\psi \sqsubseteq_s \psi' \quad \Leftrightarrow \quad (\psi \Leftrightarrow \psi') \vee \psi \equiv \bot_S \vee \psi' \equiv \top_S$$

The elements $\top_S$ and $\bot_S$ can be identified with the formulas for true and false, $\top$ and $\bot$ respectively.

The transfer function depends on the two predicates *init* and *error* that mark the abstract states as initial states or error states. For every state ψ in the set of abstract states E and every action τ out of the set of actions T the transfer relation will allow the four possible boolean combinations of these two predicates as successors. More formally, the following holds:

$$\begin{aligned} \forall \tau \in T, \psi \in E: \quad & \psi \overset{\tau}{\rightsquigarrow_s} (\mathit{init} \wedge \mathit{error}) \quad \wedge \quad \psi \overset{\tau}{\rightsquigarrow_s} (\mathit{init} \wedge \neg \mathit{error}) \quad \wedge \\ & \psi \overset{\tau}{\rightsquigarrow_s} (\neg \mathit{init} \wedge \mathit{error}) \quad \wedge \quad \psi \overset{\tau}{\rightsquigarrow_s} (\neg \mathit{init} \wedge \neg \mathit{error}) \end{aligned}$$

The merge operator checks its two arguments ψ, ψ' for subsumption:

$$\mathsf{merge}_s(\psi, \psi')) = \begin{cases} \psi \vee \psi' & \text{if } \psi \sqsubseteq_s \psi' \\ \psi' & \text{otherwise} \end{cases}$$

Here again, the merge operator will have no immediate effect in the CPA* algorithm since it will always return the same lattice element as its second argument. The CPA $\mathbb{D}_{ARG}$ that is responsible for constructing the ARG however may merge the two corresponding nodes if the subsumption holds for their wrapped abstract states.

The stop operator returns *true* if the subsumption holds for any element in reached:

$$\mathsf{stop}_s((l, \psi), \mathsf{reached}) = \begin{cases} \mathit{true} & \text{if } \exists \psi' \in \mathsf{reached} : \psi \sqsubseteq_s \psi' \\ \mathit{false} & \text{otherwise} \end{cases}$$

Refinement In SLAB, the target locations are only given implicitly in form of the error predicate. Therefore isTargetState in principle needs to check whether $\psi \wedge \mathit{error}$ is satisfiable. But since the state formulas in the initial abstract model either contain *error* or $\neg$*error* and the refinement only adds predicates, the target

state information can be determined more efficiently by caching this information for each state. Target states always split into target states and all other states never become target states.

The refinement function refine_{SliAbs}(cf. Algorithm 4) can also be used for SLAB with minor extensions. For the generation of the path formulas of the error path and also for slicing, we need to use the full transition formulas instead of just the data transition formulas. This will add the neccessary information about the program counter symbolically. In order to get a behavior similar to the original design of SLAB, we have to make sure that sliceEdges considers each possible transition τ between two states separately.

For Kojak, it is always the case that there is not more than one $\tau \in T$ that connects two states, while for SLAB there can be arbitrary many. For clarity, we call the set of direct transitions between two states in the abstract model an *edge set*. When constructing the path formula for slicing, we could simply make a disjunction of the transition formulas of all transitions in the edge set between two states. This would then lead to removing the connection between two states only if there exists no $\tau \in T$ for which the transition is possible.

By breaking this path formula down into multiple formulas, we get smaller formulas for which the satisfiability may also be solved faster. If then all but one of these formulas are unsatisfiable, we can make use of this information by removing the corresponding transitions from the edge set. Only when the edge set is empty, there is no direct connection between the two states in the ARG anymore.

The analysis for SLAB described up to this point is already functional. Whenever a spurious error path is encountered, the CEGAR refinement will discover new predicates and split the states accordingly. These new predicates may contain assumptions about the program counter values. Thus the control flow will become apparent if it is necessary for proving the program correct.

3.2.2 With Flexible-Block Encoding

The original version of SLAB uses transition bypassing as an optimization. This leads to composition of sequential transitions into larger blocks wherever possible. The similarity to LBE is evident. We can now add transition bypassing to the SLAB analysis in a similar manner as we added ABE to Kojak. This results in flexible-block encoding(FBE)[2].

As for ABE, we differentiate between abstraction and non-abstraction nodes. In the initial abstract model, each node is an abstraction node. After each refinement iteration, there might be some nodes for which transition bypassing is possible, that is:

- the node is an abstraction node
- the node has only one incoming edge with exactly one transition in the edge set
- the node has only one outgoing edge with exactly one transition in the edge set

[2]Originally we used the name dynamic-block encoding for this, but a recent publication uses that term for a conceptually different approach [24]. In order to avoid confusion we switched the name to FBE.

- the node has no self loop (neither direct or via a segment edge)

We can formalize these criteria into a block operator blk_{simple} that, given a node in the abstract model, determines whether it shall be an abstraction node or not.

For adding flexible-block encoding, we modify refine_{SliAbs} as can be seen in Algorithm 5. At the end of the while loop, we iterate through every abstraction node in the abstract model and determine whether we can convert it into a non-abstraction node using the block operator. Because this may change the structure of segment edges between abstraction nodes, as a second step we try to slice each affected segment edge. This in return might change the result of the block operator for some abstraction nodes. As a consequence both steps are repeated in a fixed-point iteration until no more nodes are converted into non-abstraction nodes. In the end, the reached set is recalculated as it might also have changed.

Algorithm 5 refinement refine_{SliAbs} with flexible-block encoding

Input: a list of abstract states waitlist,
a list of abstract states reached
Output: **false** if abstract error state reachable, **true** otherwise
Variables: a map interpolants $\subseteq E \to \mathcal{F}(X)$ of abstract states to formulas,
a list errorPath of abstract states,
a list of abstract states currentReached,
a boolean fixpoint indicating whether FBE has reached a fixed point

```
1:  currentReached = reached
2:  while ∃e ∈ currentReached : isTargetState(e) do
3:    errorPath = getErrorPath(reached)
4:    if sat(errorPath) then
5:      return (waitlist, currentReached)
6:    interpolants = getInterpolants(errorPath)
7:    for all (n, I) ∈ interpolants do
8:      split(n, I)
9:    currentReached = updateReachedSet(currentReached)
10:   for all n ∈ currentReached do
11:     sliceEdges(n)
12:   fixpoint = false
13:   while !(fixpoint) do
14:     fixpoint = performFBE(currentReached)
15:     sliceAllEdges()
16:   currentReached = updateReachedSet(currentReached)
17: return (waitlist, currentReached)
```

The resulting block encoding from performing FBE with the block operator blk_{simple} described above is very basic, as it only generates blocks with linear sequences of transitions. In LBE also simple control-flow branchings and merges are summarized into one block, as long as no loop is formed.

By changing the block operator, this can also be achieved for FBE. The new block operator blk_{large} will also return true if there are multiple incoming or outgoing edges, provided that there is only one transition in each corresponding

edge set. An important point here is that butterfly-like abstract models (cf. 3.1.2) might arise if FBE is performed with this new block operator. That means that the slicing step needs to be able to handle this, e. g. by *copy-on-slice*.

3.2.3 Interprocedural Analysis

Just like in the case of Kojak, the SLAB-like analysis as described here is inherently imprecise with regard to interprocedural programs. For Kojak we were able to fix this by adding a CPA that tracks the call stack. The information about the right procedure exit points is then encoded into the abstract model during the execution of the CPA* algorithm.

For SLAB this is not as simple. When we add a CPA that tracks the call stack to the CPA for SLAB, the CPA* algorithm does not terminate for programs that contain procedures. That is because for every abstract state there is always a successor that results from one of the procedure-call edges and therefore has a call stack that is one element larger than the call stack of the original state. For solving this problem we need to add the information about the call graph for the call-stack CPA such that a successor for a procedure-call edge is only generated if the procedure call is possible from the context of the procedure that is on top of the call stack. With this modification, the CPA* algorithm will create an abstract state for every possible call-stack configuration and every possible combination of the predicates *init* and *error*.

The necessary call-graph information can be encoded into the CFG, i. e., each procedure-call edge contains the information about the context from within which it is called. The resulting modification to the CFG is similar to the recursive CFG used by Ultimate Kojak. The difference is that in the RCFG the information about the call context is stored in the return edges, not the call edges. This indicates that there might be a natural extension that leads to an analysis that can verify recursive programs.

3.3 Connection between Slicing Abstractions and Impact

The refinement procedure refine_{SliAbs} used in the CEGAR refinement for Kojak and SLAB can also be used for other analyses. It is closely related to the refinement procedure refine_{impact} of the CPA version of Impact [20]. In both refinements, a sequence of interpolants is generated for the error path. While Impact adds the interpolants to the states and then removes those that become infeasible, the refinement refine_{SliAbs} additionally generates a split state where the negated interpolant is added instead. In an analysis like Impact where the merge operator never merges states, the abstract model has a tree-like shape and therefore this additional split state will always be removed by the slicing. The reason for this is that the interpolant is a true invariant at this position in the abstract model, as there is only one path from the tree root to the state. The only way for the split state to not be removed is if it has a different parent from which the transition is feasible, but this is never the case because of the tree shape.

As a consequence, refine_{SliAbs} will remove all states from the abstract model that are removed by refine_{impact}. It might however also remove other subtrees of the abstract model, as Impact does not check whether the edges between the states are feasible. It is not obvious whether this is an advantage or a disadvantage.

On the one hand, the removed subtrees might contain states that are still in the waitlist. In this case refine_{SliAbs} saves us from unnecessarily exploring those states. On the other hand, there might also be states in the deleted subtrees that cover other states or would cover other states in the future. Removing them means throwing away useful information that has to be rediscovered later. We will answer the question whether refine_{SliAbs} can improve an IMPACT-style analysis later in Section 4.6.

4 Evaluation

In this section we will perform benchmarks in order to evaluate how different variants of the new analyses perform compared to each other as well as compared to already existing ones.

Here is an overview of the different configurations that we will evaluate:

UKOJAK This is Kojak as implemented in the Ultimate framework [7]. We use the version that was submitted for SV-COMP18.

CPAKOJAK-ABEL CPACHECKER configured to perform an Kojak-like analysis with ABE as described in Section 3.1.2. The corresponding configuration file is `predicateAnalysis-Kojak-ABEl.properties`.

CPAKOJAK-ABEL-LIN As CPAKOJAK-ABEL, but the SMT solver uses linear instead of bit-precise theory.

CPAKOJAK-SBE The same as CPAKOJAK-ABEL, but the block operator sets every state to be an abstraction state. This results in SBE. The corresponding configuration file is `predicateAnalysis-Kojak-SBE.properties`.

CPAKOJAK-FBE The same as CPAKOJAK-ABEL, but with additional FBE. In the beginning, the block sizes are as in LBE. During the refinement, blocks can increase in size due to FBE.

CPAPREDABS-ABEL CPACHECKER configured to perform predicate abstraction with ABE. The corresponding configuration file is `predicateAnalysis.properties`.

CPASLAB-SBE A configuration of CPACHECKER that performs a basic SLAB-like analysis as described in Section 3.2.1. Handling of the call stack has not been implemented, therefore this analysis is inherently imprecise. The corresponding configuration file is `predicateAnalysis-Slab-SBE.properties`.

CPASLAB-FBE The same as CPASLAB-SBE, but with FBE enabled (cf. Section 3.2.2).

CPAIMPACT-ABEL A configuration where CPACHECKER performs a predicate analysis with a refinement as described in "Lazy Abstraction with Interpolants" [19]. ABE is used to speed up the analysis. The corresponding configuration file is `predicateAnalysis-ImpactRefiner-ABEl.properties`.

CPAIMPACT-SBE Like CPAIMPACT-ABEL, but with SBE. The corresponding configuration file is `predicateAnalysis-ImpactRefiner-SBE.properties`.

CPASliAbs-abel Like the configuration for Impact, but the the refiner of impact is replaced by refine_{SliAbs}. The corresponding configuration file is `predicateAnalysis-ImpactRefiner-ABEl.properties`.

CPASliAbs-sbe This configuration uses SBE instead of ABE. Everything else is identical to CPASliAbs-abel. The corresponding configuration file is `predicateAnalysis-ImpactRefiner-SBE.properties`.

In the configurations that work with FBE (CPAKojak-fbe and CPASlab-fbe), the basic block operator blk_{simple} is used. This is only because the current implementation does not yet support *copy-on-slice* (cf. 3.2.2).

Whenever ABE is used, the block operator is configured to create abstraction states at loop heads.

If not stated otherwise, MathSAT5 with bit-precise representation is used in all configurations of CPAchecker. The version of CPAchecker used is revision 28465.

4.1 Benchmark Overview

Tasks In order to perform a meaningful benchmark, we need a set of tasks (C programs) that is both large enough and representative for the kinds of problems that verification has to deal with.

For benchmarking, we choose a well-established set of tasks from the Competition on Software Verification(SV-COMP). While the benchmark set of SV-COMP has many categories, we will restrict our benchmark to the category reach safety, which contains 2942 tasks in 10 subcategories (see Table 1). Our analyses will work for some of the other categories as well, but reach safety should already be enough to compare the basic performance of different configurations. The version of the tasks we will use is the one at the svcomp18 tag in the official benchmark repository[3].

Environment It has been shown that for reliable benchmarking several aspects have to be considered that can easily be done wrong [21]. For example, the exact measurement of resource consumption as well as resource limitation are not trivial. This can be done by making use of process namespaces and cgroups, features that are available in the linux kernel since version 3.16. We will use BenchExec[4] to execute the tasks, which creates a containerized environment for each task to run in. For the benchmark systems we use machines with uniform hardware. Each machine consists of an Intel Xeon E3-1230 v5 CPU with 8 processing units and 33 GB of memory. The operating system on all machines is Ubuntu 16.04, with the long-term support linux kernel version 4.4.0.

With BenchExec, we are able to set a time limit of 900 seconds, a memory limit of 15 GB and ensure that the tool can use all 8 processing units. Note that this is the same environment that was used for SV-COMP18, thus the results should be directly comparable to the results for any tool used in the competition. While the kernel version used in SV-COMP18 also was 4.4.0-101, we use version 4.4.0-128 which includes patches that mitigate recently discovered vulnerabilities in the Intel x86 CPU hardware architecture [25, 26]. As these might have a measurable

[3] `https://github.com/sosy-lab/sv-benchmarks/tree/svcomp18`
[4] https://github.com/sosy-lab/benchexec

influence on the computation times [27], we will not reuse the results of UKojak from SV-COMP18 and instead reproduce them with the current kernel version.

Results An overview of the benchmark results for each of the configurations explained in Section 4 is shown in Table 1.

4.2 Comparision with Ultimate Kojak

In order to assess the validity of our implementation, we can compare the new Kojak implementation for CPAchecker with the original version UKojak. As UKojak uses LBE, it is best compared to CPAKojak-abel. Another important component is the SMT solver that is used for satisfiability checks as well as the underlying theory for integer and floating point representation. According to the technical paper [7], UKojak uses Z3 for feasibility checks of error paths and transition formulas and does interpolation via SMTInterpol. It is unclear whether this still holds for the version of UKojak submitted to SVCOMP18 or whether Z3 has been dropped in favor of SMTInterpol.

The default solver for CPAchecker is MathSAT5 with bit-precise representation of integers and floats. We can change this to a linear representation (floats are then encoded as rational fractions) which results in the configuration CPAKojak-abel-lin. Changing the solver from MathSAT5 to SMTInterpol does not have a significant impact on the benchmark results. As a consequence we do not show benchmarks where MathSAT5 is replaced by SMTInterpol.

Now we can have a look at UKojak, CPAKojak-abel and CPAKojak-abel-lin. These tool configurations are shown in the first three columns of Table 1.

Recursion The first observation is that our implementations of Kojak in CPAchecker cannot handle recursive programs, while UKojak can solve some of the tasks in this category by making use of nested word automata with nested interpolants [7, 22, 23].

Soundness Both UKojak and CPAKojak-abel are sound, as there are (close to) no incorrect results. The one incorrect result of CPAKojak-abel in the heap subcategory also appears for other sound analyses of CPAchecker like CPAPredAbs-abel and is therefore not a unsoundness that originates from our particular implementation.

While CPAKojak-abel-lin manages to solve more tasks than CPAKojak-abel, it also produces incorrect results. This is as expected, since the linear representation is faster but also unsound.

Performance In general, the number of tasks solved per subcategory are similar for CPAKojak-abel and UKojak. Overall, CPAKojak-abel can classify more tasks correctly.

In subcategories where the numbers of solved tasks are more or less the same, one would expect the tasks to be the same for both tools. This is however not always the case, indicating that there are differences in other parts of the tools that are not related to the Kojak refinement algorithm.

A scatter plot of the CPU time comparing both tools is shown in Fig. 10. Only tasks that both tools classified correctly are drawn. Points below the central diagonal are solved faster by CPAKojak-abel while for the points above UKojak is

	UKojak	CPAKojak-abel-lin	CPAKojak-abel	CPAKojak-sbe	CPAKojak-fbe	CPAImpact-abel	CPAImpact-sbe	CPAPredAbs-abel	CPAPredAbs-sbe	CPASliAbs-abel	CPASliAbs-sbe	CPASlab-sbe	CPASlab-fbe
arrays (167 tasks)													
correct results	10	11	4	3	5	4	3	6	6	3	3	7	5
correct true	6	3	2	2	3	2	2	3	2	1	2	2	2
correct false	4	8	2	1	2	2	1	3	4	2	1	5	3
incorrect results	0	16	0	0	0	0	0	0	0	0	0	19	13
bitvectors (50 tasks)													
correct results	19	19	36	27	33	34	30	39	28	31	30	17	19
correct true	11	12	25	17	23	23	23	29	20	20	23	12	14
correct false	8	7	11	10	10	11	7	10	8	11	7	5	5
incorrect results	0	27	0	0	0	0	1	0	0	0	1	4	5
control flow (94 tasks)													
correct results	31	68	59	44	52	55	24	62	34	39	23	4	8
correct true	16	33	29	16	26	26	11	33	14	17	10	2	6
correct false	15	35	30	28	26	29	13	29	20	22	13	2	2
incorrect results	0	0	0	0	0	0	0	0	0	0	0	0	0
ECA (1149 tasks)													
correct results	328	235	200	6	201	450	3	477	4	241	3	1	3
correct true	267	151	160	4	162	301	3	333	3	167	3	1	3
correct false	61	84	40	2	39	149	0	144	1	74	0	0	0
incorrect results	0	93	0	0	0	0	0	0	0	0	0	0	0
floats (172 tasks)													
correct results	33	53	34	6	34	28	6	91	11	27	5	2	2
correct true	29	34	8	2	8	2	2	65	7	1	1	2	2
correct false	4	19	26	4	26	26	4	26	4	26	4	0	0
incorrect results	0	66	0	0	0	0	0	1	0	0	0	0	0
heap (181 tasks)													
correct results	100	117	108	94	103	110	94	114	94	100	94	54	77
correct true	52	71	62	51	61	64	52	66	54	59	52	35	43
correct false	48	46	46	43	42	46	42	48	40	41	42	19	34
incorrect results	0	4	1	1	1	1	1	1	1	1	1	1	3
loops (163 tasks)													
correct results	96	103	76	73	74	74	71	82	69	71	68	64	63
correct true	60	68	45	41	45	41	40	47	38	40	37	34	33
correct false	36	35	31	32	29	33	31	35	31	31	31	30	30
incorrect results	0	6	0	0	0	0	0	0	0	0	0	7	7
product lines (597 tasks)													
correct results	295	462	463	327	463	578	309	551	406	203	304	0	78
correct true	204	275	275	231	275	313	155	317	228	77	150	0	76
correct false	91	187	188	96	188	265	154	234	178	126	154	0	2
incorrect results	0	0	0	0	0	0	0	0	0	0	0	0	0
recursive (96 tasks)													
correct results	45	0	0	0	0	0	0	0	0	0	0	0	0
correct true	19	0	0	0	0	0	0	0	0	0	0	0	0
correct false	26	0	0	0	0	0	0	0	0	0	0	0	0
incorrect results	0	0	0	0	0	0	0	0	0	0	0	0	0
sequentialized (273 tasks)													
correct results	17	155	131	12	131	127	22	116	14	109	23	1	2
correct true	0	33	17	4	20	13	3	22	3	9	3	0	0
correct false	17	122	114	8	111	114	19	94	11	100	20	1	2
incorrect results	0	14	0	0	0	0	0	0	0	0	0	0	1
total (2942)													
correct results	974	1223	1111	592	1096	1460	562	1538	666	824	553	150	257
correct true	664	680	623	368	623	785	291	915	369	391	281	88	179
correct false	310	543	488	224	473	675	271	623	297	433	272	62	78
incorrect results	0	226	1	1	1	1	2	2	1	1	2	31	29

Table 1: Benchmark results for the reach safety tasks when run with different tool configurations

faster. Points that are between the other two diagonals do not deviate by more than a factor of 10 in their CPU time.

There is a cluster of tasks that CPAKojak-abel can solve in less than 20 seconds, In this cluster, CPAKojak-abel tends to be faster than UKojak. In the other cluster of

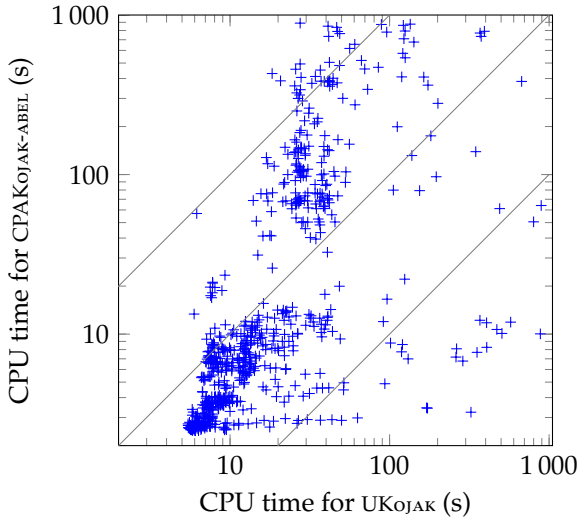


Figure 10: Scatter plot comparing CPU time of UKojak and CPAKojak-abel for each task that both tools where able to label correctly

task for which CPAKojak-abel takes more than 20 seconds, UKojak tends to be faster. This indicates that our implementation does not scale as good with problem size as UKojak.

There are two possible explanations for this. For one, the implementation of block encoding via ABE has a certain overhead, especially for Kojak. When a segment edge is split, all non-abstraction states have to be copied. Every time the edge formula for a segment edge is needed – either as part of the error path or when slicing – it has to be constructed using the current configuration of the reachability graph. The second explanation is the difference in the time that it takes solving the SMT queries. This depends on the solver as well as on the underlying theory.

Fig. 11 shows a scatter plot between CPAKojak-abel to CPAKojak-abel-lin. While the linear theory tends to decrease the CPU time, it is not enough to change the scatter plot in Fig. 10 significantly. Thus the second explanation can not explain the cluster of tasks for which CPAKojak-abel takes longer than UKojak.

There is also strong evidence that supports the first explanation. The majority of tasks in the cluster where CPAKojak-abel takes more than 20 seconds are from the ECA subcategory. The control flow of the tasks in this category consist of large sections of acyclic control-flow branchings and merges. This is exactely the kind of tasks for which we expect the overhead of the block encoding used in CPAKojak-abel to be largest.

4.3 Effect of Block Encoding on Kojak

We still need to determine whether ABE has the desired effect on performance for our implementation of Kojak. We can also enable FBE for Kojak that was originally added for SLAB and evalulate whether this further improves the analysis.

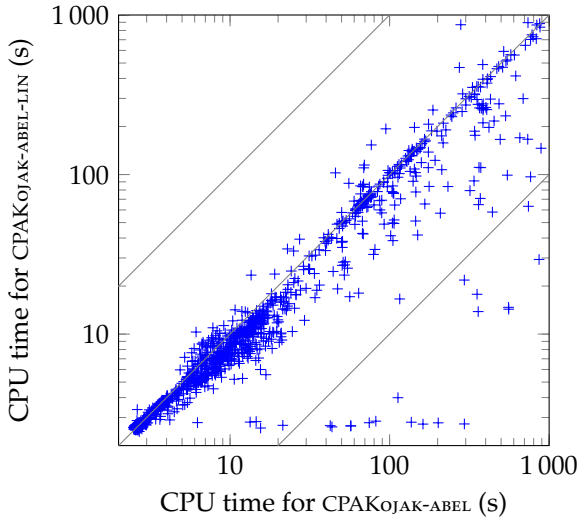


Figure 11: Scatter plot comparing CPU time of CPAKojak-abel and CPAKojak-abel-lin for each task that both tools were able to label correctly

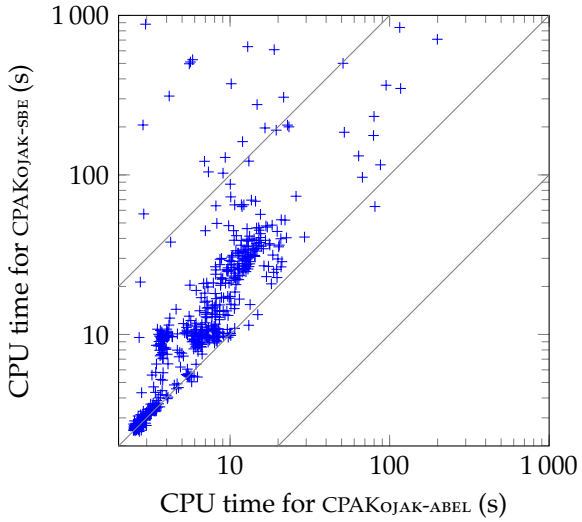


Figure 12: Scatter plot comparing CPU time of CPAKojak-abel and CPAKojak-sbe for each task that both tools were able to label correctly

4.3.1 Effect of ABE

Fig. 12 shows a scatter plot comparing the SBE version with the ABE version of CPAKojak. ABE almost always leads to a reduction of computation time when compared to SBE. There is also a significant number of tasks that only the ABE version is able to solve. This cannot be seen in the scatter plot, but becomes clear from Table 1.

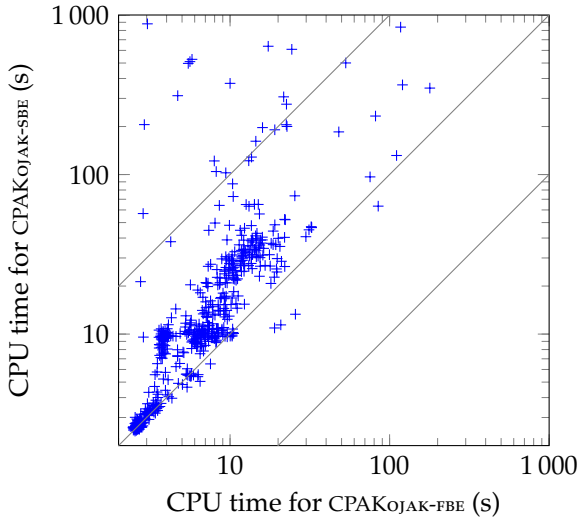


Figure 13: Scatter plot comparing CPU time of CPAKojak-fbe and CPAKojak-abel for each task that both tools were able to label correctly

4.3.2 Effect of FBE

Since ABE has a positive effect on Kojak, we can ask the question whether FBE can further increase the performance. Fig. 15 shows a quantile plot containing the benchmark results for CPAKojak-abel and CPAKojak-fbe. It becomes very clear from the plot that FBE does add a minor overhead to the analysis and does not in any way increase performance. Keep in mind though that the block operator for FBE used here is still blk_{simple} because of limitations of the current implementation. The results could change of the stronger block operator blk_{large} is used.

4.4 Comparison of SLAB and Kojak

As can be seen in Table 1, our implementation of SLAB cannot compete with any of the other analyses. It also has a high number of false alerts which arise from the fact that our implementation does not yet treat the call stack correctly. The main reason for the performance loss when compared to Kojak is due to the high number of solver calls. A comparison is shown in Fig. 14. In the first iterations of the refinement, the reason for the infeasibility of the error path is usually a mismatch in program counter value. The program counter value is then added to one state via splitting and the size of the remaining self-loop edge set for the split state usually decreases by one (cf. Fig. 8). If the complete control-flow information is unrolled in this manner, it typically leads to a number of slicing operations that is quadratic in the number of transitions in the CFG. As a consequence, the number of solver calls required for rediscovering the complete control flow can reach in the millions for bigger programs. In the benchmark the time limit of 900 s is always exceeded before a million solver calls are reached. This indicates that a solver call takes at least 1 ms on average.

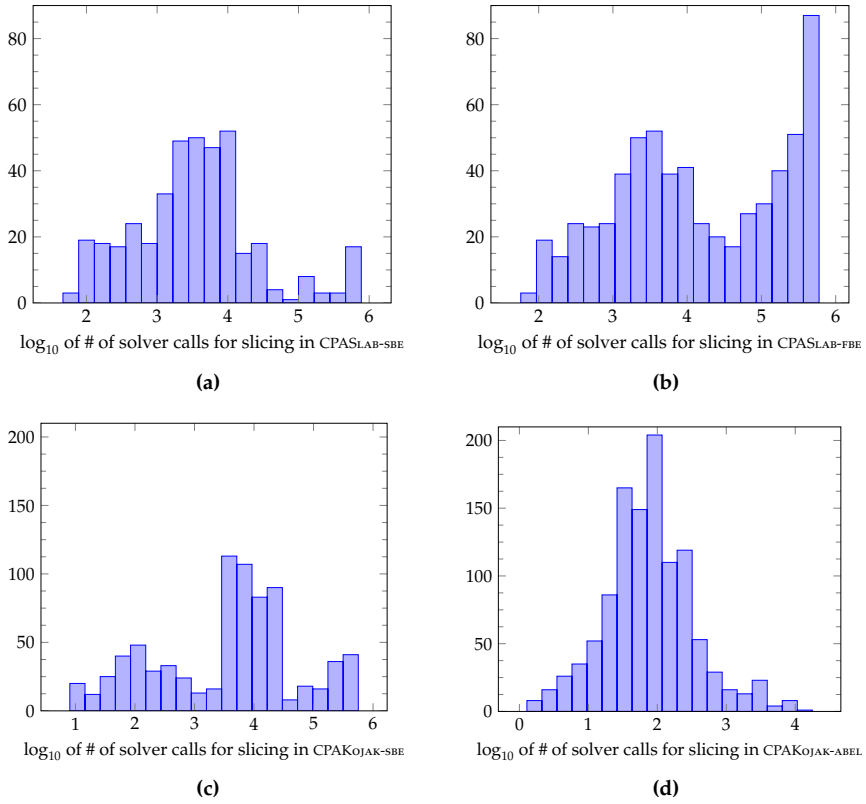


Figure 14: Logarithmic histograms of the number of solver calls performed by **(a)** CPASlab-sbe, **(b)** CPASlab-fbe,**(c)** CPAKojak-sbe and **(d)** CPAKojak-abel. This includes all tasks where this statistic is available, even if the computed result is incorrect or the task did not finish in time.

4.5 Effect of FBE on SLAB

4.6 Kojak vs. Predicate Abstraction vs. Impact

4.6.1 With LBE or ABE

Fig. 15 shows a quantile plot comparing UKojak, CPAKojak-abel, CPAPredAbs-abel, Impact, and CPASliAbs-abel. All analyses shown use ABE or LBE. While our implementation of Kojak performs better than UKojak, it falls short of predicate abstraction and Impact analyses. This is despite the fact that they are very similar: they all use the predicates from a sequence of interpolants for an infeasible error path to refine their abstract model.

While CPAPredAbs-abel and Impact can use this information to decide whether to remove states, the Kojak-based analyses need to perform separate solver calls in order to decide which edges can be sliced. States are then removed only as a result of this slicing whenever they get disconnected from the initial state. When an edge cannot be sliced, it can get duplicated in the next refinement iteration. As a result, the number of edges that need to be checked for slicing

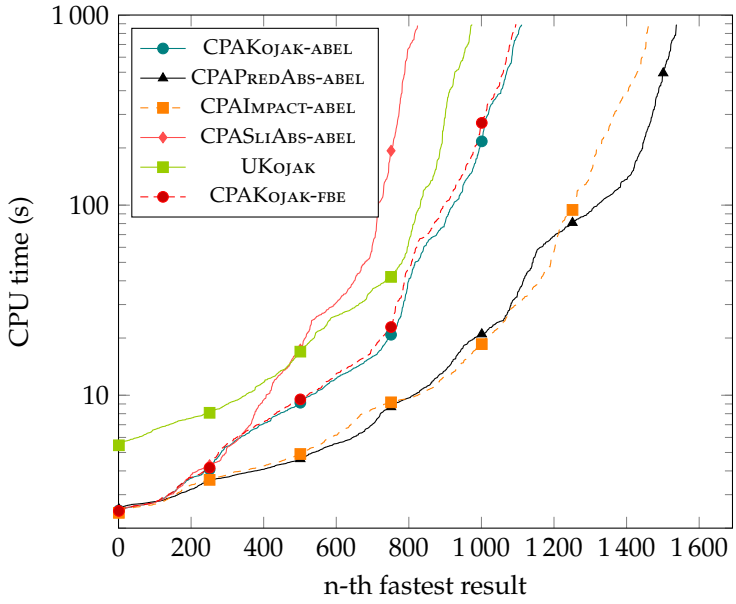


Figure 15: Quantile plot comparing CPAKojak-abel, CPAPredAbs-abel, Impact, CPASliAbs-abel and UKojak. All configurations use either LBE or ABE configured to be compareable to LBE.

can duplicate. This overhead of slicing could explain the performance gap of Kojak when compared to CPAPredAbs-abel and Impact.

Because of this, we try to optimize the slicing as much as possible in order to rule out that the performance gap is simply a result of missing optimizations. For some edges we can decide immediately that they are infeasible because of the structure of the interpolants [6]. We slice these edges without the need of a time-consuming solver call and also only attempt to slice an edge when the state formula of one of the states it connects has changed.

Fig. 15 also shows the performance of CPASliAbs-abel, a configuration that is like CPAImpact-abel except the refinement refine_{impact} is replaced by the slicing refinement refine_{SliAbs}. As discussed previously in Section 3.3 these analyses are very similar. refine_{SliAbs} removes more parts of the abstract model than refine_{impact} because it also checks the edges of the changed states for infeasibility. These slicing checks get more expensive if the segment size is increased,e. g., if ABE is used. This explains the gap between CPASliAbs-abel and CPAImpact-abel.

4.6.2 SBE

We already saw in Section 4.3.1 that ABE has a positive impact on the performance of our Kojak implementation. It is also clear that the same holds for predicate abstraction and Impact [20]. What is however still missing is a comparison of the SBE variants of the three tools. After all, the disadvantage of CPAKojak-abel arose from the fact that slicing of large segments has a high cost of computation time. When the block size is reduced to SBE, this effect should be reduced to some extend.

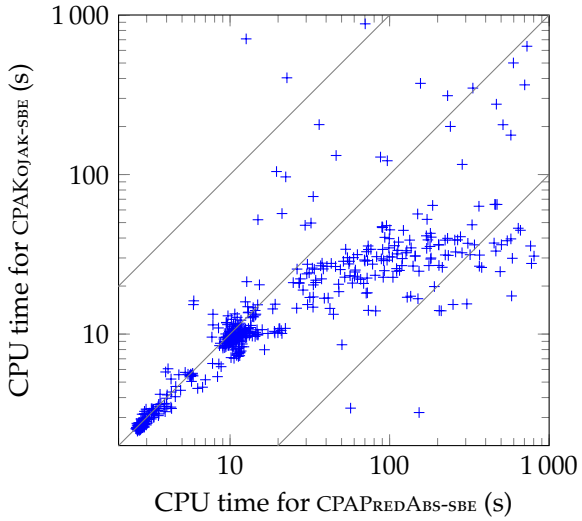


Figure 16: Scatter plot comparing CPU time of CPAPredAbs-sbe and CPAKojak-sbe for each task that both tools were able to label correctly

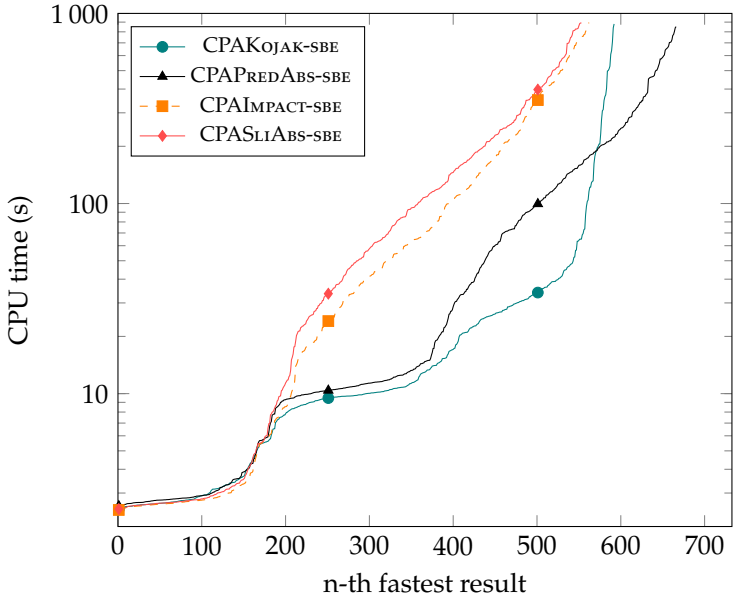


Figure 17: Quantile plot comparing CPAKojak-sbe with CPAPredAbs-sbe and CPAImpact-sbe. All configurations use SBE

Fig. 17 shows a quantile plot comparing the SBE versions of the tools under consideration. Surprisingly, CPAKojak-sbe is even faster than CPAPredAbs-abel for most of the tasks that were solved correctly. CPAPredAbs-sbe however still manages to solve more tasks. As can be seen in Table 1, this is mainly due to the product-lines subcategory. In most of the other subcategories, the number of correct results for CPAKojak-sbe and CPAPredAbs-sbe are remarkably close.

5 Conclusion

In this section we summarize our findings and give an overview of possible future work that can be done on the two abstraction-slicing analyses.

5.1 Summary

We were able to successfully implement basic versions of Kojak and SLAB into the CPAchecker framework. Both analyses share a large fraction of their implementation and individual components of the implementation like the refiner are also available for use by other analyses. In the benchmark we performed, our implementation of Kojak is able to compete with UKojak.

The usage of ABE instead of LBE has a measurable overhead for programs with very large block sizes. When using SBE, our implementation of Kojak is faster than predicate abstraction for most tasks, though it does not manage to solve more tasks in total.

By switching to ABE this advantage in speed vanishes. This indicates that increasing the block size comes with a drawback for Kojak, since the cost for slicing edges is increased. As predicate abstraction does not slice edges, it does not suffer from the same drawback. Apart from this drawback, we were also able to show that the refinement used for abstraction slicing is very similar to the refinement used in Impact.

For SLAB, we have shown that an optimization similar to ABE that we call FBE can be used to increase the performance of the analysis. FBE has however no positive effect on analyses that can already use ABE.

A comparison of SLAB with Kojak shows that usage of control-flow information is preferable to treating the program counter symbolically.

5.2 Prospects

While a lot of work already has been done, there are still a lot of open points for improvement.

One of them is that we did not yet consider recursive programs. Since a way how this can be achieved is already known for UKojak, it should not be too complicated to add this to our implementation of Kojak and potentially also to SLAB. This might also naturally lead to elimination of function inlining, which could also give a performance boost to nonrecursive program handling.

In our evaluation, we were only able to use a very basic version of FBE due to missing support of the current implementation for a more general block operator. In order to be sure that FBE has no positive effect on analyses that can already use ABE, the corresponding experiments should be repeated as soon as the stronger block operator is supported.

Regarding our implementation of SLAB, the original tool also used a number of additional optimization which we did not yet implement. Maybe adding these or finding a new optimization can speed up our SLAB implementation significantly without effectively turning it into a variant of Kojak.

Up to now we did not yet consider different interpolation methods. CPAchecker enables us to vary a wide number of building blocks like the interpolation method independently. A systematic evaluation of the effects of changing these building blocks might give interesting insights. As Kojak is able to generate noninductive

invariants, adding support of witness generation and validation to our analysis is also one of the logical next steps.

Bibliography

[1] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*, pages 504–518. Springer, 2007.

[2] Philipp Wendler. Towards practical predicate analysis. PhD Thesis, University of Passau, Software Systems Lab, 2017.

[3] Henny B. Sipma, Tomás E Uribe, and Zohar Manna. Deductive model checking. In *International Conference on Computer Aided Verification*, pages 208–219. Springer, 1996.

[4] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. Slicing abstractions. *Fundam. Inf.*, 89(4):369–392, December 2008.

[5] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. Slab: A certifying model checker for infinite-state concurrent systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 271–274. Springer, 2010.

[6] Evren Ermis, Jochen Hoenicke, and Andreas Podelski. Splitting via interpolants. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 186–201. Springer, 2012.

[7] Evren Ermis, Alexander Nutz, Daniel Dietsch, Jochen Hoenicke, and Andreas Podelski. Ultimate kojak. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 421–423. Springer, 2014.

[8] Ákos Hajdu, Tamás Tóth, András Vörös, and István Majzik. A configurable cegar framework with interpolation-based refinements. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 158–174. Springer, 2016.

[9] Tamás Tóth, Akos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. *Formal Methods in Computer-Aided Design FMCAD 2017*, page 176, 2017.

[10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010, Lugano, October 20-23)*, pages 189–197. FMCAD, 2010.

[11] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 25–32. IEEE, 2009.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[13] Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on SMT-based software verification. *Journal of Automated Reasoning*, 2017.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[15] Hassen Saïdi. Model checking guided abstraction and analysis. In *International Static Analysis Symposium*, pages 377–396. Springer, 2000.

[16] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[17] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 37(1):58–70, 2002.

[18] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[19] Kenneth L. McMillan. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*, pages 123–136. Springer, 2006.

[20] Dirk Beyer and Philipp Wendler. Algorithms for software model checking: Predicate abstraction vs. IMPACT. In Gianpiero Cabodi and Satnam Singh, editors, *Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2012, Cambrige, UK, October 22-25)*, pages 106–113. FMCAD, 2012.

[21] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, pages 1–29, 2017.

[22] Daniel Dietsch. *Automated Verification of System Requirements and Software Specifications*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2016.

[23] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. *ACM Sigplan Notices*, 45(1):471–482, 2010.

[24] Daniel Dietsch, Marius Greitschus, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, Andreas Podelski, Christian Schilling, and Tanja Schindler. Ultimate taipan with dynamic block encoding. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 452–456. Springer, 2018.

[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[27] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Robert L. DeLeon, and Thomas R. Furlani. Effect of meltdown and spectre patches on the performance of HPC applications. *CoRR*, abs/1801.04329, 2018.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher weder einer anderen Prüfungsbehörde vorgelegt noch veröffentlicht.

München, 2018-06-26 _____

Martin Spießl