UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Master Thesis

in Computer Science

Software Verification based on Adjustable Large-Block Encoding

Philipp Wendler

Supervisor:

Prof. Dr. Dirk Beyer

March 26, 2010

Abstract

Predicate abstraction is a widely used technique for software model checking. Most approaches employ counterexample-guided abstraction refinement (CEGAR). The predicates needed for the abstraction of the program states are found through the generation of Craig interpolants for infeasible paths. Traditionally, the abstractions were computed after each statement of the program, leading to a large number of costly abstraction computations. A faster approach was proposed which summarizes larger parts of a program into one formula, and computes the abstraction only once for a whole block of statements. This is called Large-Block Encoding (LBE), the previous method being called Single-Block Encoding (SBE), respectively. LBE uses blocks that are always as large as they can be without including function calls and loops. In this work, an extended analysis is presented which is more flexible and allows the use of block sizes ranging from SBE to LBE and beyond, through the adjustment of several parameters. Such a unification of different concepts makes it easier to understand and analyze the fundamental properties of the analysis, and makes the differences of the variants more explicit. Certain configurations could not be considered before in experiments, because the tool implementations only allowed for configurations that are the extreme cases of the new unified formulation. Benchmarks on example C programs are reported with different configurations in order to identify one that is generally the fastest.

Contents

List of Figures

List of Tables

1 Introduction

The verification of software products is very important, as our world depends more and more on the correct functioning of software in all possible areas. Erroneous programs can even lead to deadly accidents [Neu09]. Several methods for finding bugs and checking safety properties are based on static analysis, i.e., the analysis of the program source code without actually executing the program. A comparison can be found in [DKW08]. The best precision is provided by software model checking [JM09], which verifies that a program conforms to a given specification. Such a specification could, for example, be given by the programmer, by specifying predicates that must always hold at a certain program location (e.g., through the use of assertions). Model checking with predicate abstraction [GS97] has made a lot of progress, being implemented in several tools like SLAM [BR02] and BLAST [BHJM07], which are able to verify properties of C programs. These tools use counterexample-guided abstraction refinement [CGJ$^+$03] and lazy abstraction [HJMS02], automatically finding the necessary predicates through generating Craig interpolants [Cra57] for spurious counterexamples [HJMM04]. However, their use for verifying real software products is still limited because of the large amount of resources (especially time) needed to verify bigger programs. Large-Block Encoding (LBE) [BCG$^+$09] has been proposed as a way to speed up the analysis based on predicate abstraction of larger program blocks at a time. The previous approach, called Single-Block Encoding (SBE), is slower because it needs a costly abstraction computation after each program statement. In this work, an extension to LBE called Adjustable Large-Block Encoding is presented that allows configuration of the block size in a wide range. Just by choosing the values of a few parameters not only SBE and LBE can be used, but also block sizes between these two configurations and beyond LBE are possible. This has been implemented in CPACHECKER [BK09], a tool for static analysis that implements the Configurable Software Verification [BHT07] framework and can be used as a software model checker. This implementation was used to experiment with different configurations, benchmarking the time needed to verify several example C programs and evaluating the results.

1.1 Related Work

There are several model checkers that use the traditional SBE approach together with counterexample-guided abstraction refinement (CEGAR) [CGJ+03], including Slam [BR02] and Blast [BHJM07]. The tool SatAbs [CKSY05] is also based on CEGAR but uses symbolic search in the abstract space instead of abstractions. More similar to LBE is the work of McMillan [McM06], which also follows the lazy-abstraction paradigm [HJMS02], but does not use abstractions. Instead, it directly adds the predicates extracted from the Craig interpolants [Cra57] [HJMM04] to the abstract states. Bounded Model Checking [BCCZ99], for example implemented in CBMC [CKL04], focuses on efficiently finding bugs in programs, but is not precise enough to verify that a program conforms to a specification.

1.2 Structure

Chapter 2 provides the necessary background, including the definition of predicate abstraction, counterexample-guided abstraction refinement and Large-Block Encoding. The CPA framework, which is used to encode the analysis, is also introduced here. The contributions of this work are presented in Chapter 3 and its implementation is described in Chapter 4. The experiments conducted to evaluate the new approach and the discussion of the results can be found in Chapter 5, with some resulting ideas for future work outlined in Chapter 6.

2 Background

2.1 Input Language and Control-Flow Automata

The language of the analyzed programs is a simple imperative programming language containing loops, conditional branches and unconditional jumps (goto). The only type of variables ranges over integers and is assumed to be of unconstrained precision (no overflows). References or pointers, as well as arrays, are not part of the language. All instructions are either simple side-effect-free assignments or assume operations.

Control-flow automata (CFA) [BCG+09] are used to encode programs given in this language. A CFA $A = (L, G)$ consists of a set L of program locations, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, where Ops is the set of all possible operations supported by the input language. The locations model the program counter l. Each control-flow edge $g \in G$ contains an operation that is executed when the control flows from the predecessor location to the successor location of this edge. The set of all variables that occur in the operations from the edges of A is denoted by X. A *program* $P = (A, l_0)$ consists of a CFA A and a designated node from L, the start location l_0.

An example program written in a C-like syntax can be seen in Figure 2.1. It acquires and frees a lock repeatedly and contains several checks verifying the correct state of the lock. Figure 2.2 contains the CFA for this program, with node 0 being the start location l_0.

A *concrete data state* of a program is a set $c : X \to \mathbb{Z}$ of mappings that assigns a value to each variable of a program. The set of all such states is called $\mathcal{C}$. Any subset of C is called a *region*. A *concrete state* $(c, l) \in \mathcal{C} \times L$ is a variable assignment at a specific program location. A *concrete path* is a finite sequence $\langle (c_1, l_1), \ldots, (c_n, l_n) \rangle$ of states such that $\forall i \in \{1, \ldots, n-1\} : (l_i, op_i, l_{i+1}) \in G$. Concrete paths are *feasible* if each concrete state c_i on the path is produced from its predecessor c_{i-1} by modifying c_{i-1} according to the operation op_{i-1} on the edge between them, that is, if it is possible that the program would follow this path if it was executed.

```
int f(int p, int n) {
  int LOCK ;
  LOCK = 0;

  while (n >= 0) {
    if (p) {
      if (LOCK) {
        goto ERROR;
      }
      LOCK = 1;
    }

    if (p) {
      if (! LOCK) {
        goto ERROR;
      }
      LOCK = 0;
    }

    n--;
  }

  if (! LOCK) {
    return 0;
  }

ERROR:
  return 1;
}
```

Figure 2.1: Example program `lock-loop.c`

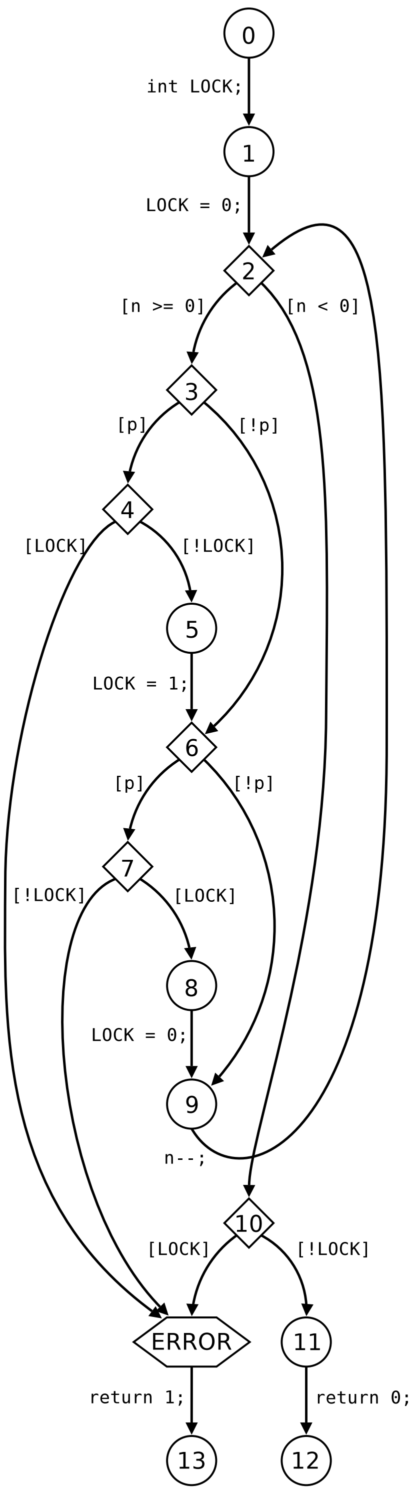


Figure 2.2: CFA for `lock-loop.c` *assume*(p) is represented by `[p]`, nodes with two outgoing edges are of rhombic shape

Given an *error location* $l_E \in L$, a program $P = (A, l_0)$ is called *l_E-safe* if there is no feasible concrete path in P starting at l_0 and ending in l_E. In the example, the error location would be the CFA node labeled `ERROR`. With software model checking, this is verified by unrolling the CFA into a tree which contains all possible concrete states and checking the feasibility of each concrete path. Programs whose specifications are given via assertions in the source code can be reduced to l_E-checking by using the following implementation of assertions:

```
assert(p) {
        if ¬p then
l_E :       exit(1);
}
```

2.2 Predicate Abstraction

When using predicate abstraction [GS97], the concrete states of a program are not stored explicitly. Instead, predicates from a quantifier-free theory $\mathcal{T}$ are used to model program states. This abstraction is not exact, but sound (i.e., it over-approximates the set of possible concrete states), possibly finding concrete paths that are not feasible, but never missing a feasible concrete path. The most commonly used theory is linear arithmetic with uninterpreted functions (LA+EUF). An efficient solver for satisfiability modulo theories (SMT) [Seb07] that supports the chosen theory has to be available. There are several approaches providing reasonably fast abstraction computations for LA+EUF [LNO06] [CCF$^+$07], implemented in tools like MathSAT [BCF$^+$08]. With $\mathcal{P}$ being a set of predicates from $\mathcal{T}$, let φ be a first-order formula over predicates from $\mathcal{P}$. The formula φ represents the region r_φ of all concrete states that imply φ, i.e., $r_\varphi = \{c \mid c \models \varphi\}$. The set of concrete states that are reachable from any of the states of a region by executing a given operation $op \in Ops$ is represented by the *strongest post-condition operator* SP_{op} [BCG$^+$09]. For an assignment operation $x := e$ with $x \in X$, a state c' is reachable from the state c if $c' = c[x \mapsto e]$, and for an assume operation $assume(p)$ with a predicate p over the variables from X, c' is reachable from c if $c' = c$ and p is true after all free variables in p have been replaced by their value from c. Thus, the strongest postcondition operator is $\mathsf{SP}_{x:=e}(\varphi) = \exists x' : \varphi[x \mapsto x'] \wedge (x = e[x \mapsto x'])$ for an assignment operation and $\mathsf{SP}_{assume(p)}(\varphi) = p \wedge \varphi$ for an assume operation.

A *program path* σ is a finite sequence $\langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ of pairs of operations from Ops and program locations from L such that $\forall i \in \{1, n\} : (l_{i-1}, op_i, l_i) \in G$

(a path is a walk through the CFA starting at l_0). The definition of the strongest postcondition operator is extended to paths by the successive application of SP_{op} for all operations in the path: $\mathsf{SP}_\sigma(\varphi) = \mathsf{SP}_{op_n}(\ldots \mathsf{SP}_{op_1}(\varphi)\ldots)$. A program path is called *feasible* if $\mathsf{SP}_\sigma(true)$ is satisfiable. Abstract reachability of a concrete state (c, l) is defined by the existence of a feasible program path σ whose final location is l and whose final region includes c (i.e., $c \models \mathsf{SP}_\sigma(true)$). Similarly, a program location l is reachable if there is any reachable state $(\cdot, l)$.

Computing the reachability of a path after every step would be too expensive computationally, because many such checks would be necessary, and the size of the involved formulas increases with the length of the path. Therefore, abstractions are computed that summarize the information known at some location of a path. Given a path $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$, an abstraction φ for $\mathsf{SP}_\sigma(true)$ should contain the information necessary to prove that a path $\sigma' = \langle(op_1, l_1), \ldots, (op_{n+1}, l_{n+1})\rangle$ is infeasible, if it indeed is. As the size of φ does not increase with the size of σ, in general the satisfiability check for $\mathsf{SP}_{op_{n+1}}(\varphi)$ should be faster than for $\mathsf{SP}_{\sigma'}(true)$, hopefully compensating for the cost of computing the abstraction. However, if the abstraction was to imprecise, the analysis could fail to prove an infeasible path σ' as infeasible and the program could be considered unsafe, even if it is in fact safe. The *precision* of an abstraction is a finite subset π of predicates from $\mathcal{P}$ that is used for computing the abstraction. For the scope of this section, this set is considered to be fixed and given externally. An abstraction for the formula φ computed under the precision π is written as φ^π. It is called an *abstract state* as it is an abstract representation of the region (which is a set of concrete states) represented by φ. Given a method to compute the abstraction for a formula φ, the abstract strongest postcondition operator is defined as $\mathsf{SP}^\pi_{op}(\varphi) = (\mathsf{SP}_{op}(\varphi))^\pi$. During analysis, this is used to compute the successor φ'^π of an abstract state φ^π, so that $\varphi'^\pi = (\mathsf{SP}_{op}(\varphi^\pi))^\pi$.

One further performance improvement can be achieved by using not one set π of predicates for all abstraction computations, but to use a function $\Pi : L \to 2^{\mathcal{P}}$. This assigns a set of predicates (a precision) to each program location. When an abstraction is computed at location l, only $\Pi(l)$ is used as the precision, and not all known predicates. This decreases the number of predicates for most of the abstractions, as normally a lot of predicates are only locally useful to prove that certain paths are infeasible, but are not necessary in other parts of the program.

For the computation of an abstraction of a formula φ under a precision π, two different methods can be used which differ in precision and performance.

2.2.1 Cartesian Predicate Abstraction

With *Cartesian predicate abstraction* [BPR01], it is checked independently for each predicate $p \in \pi$ if φ entails p. Then the Cartesian abstraction $\varphi^{\pi}_{\mathbb{C}}$ of φ under the precision π is $\varphi^{\pi}_{\mathbb{C}} = \bigwedge \{p \in \pi \mid \varphi \Rightarrow p\}$, i.e., the conjunction of all those predicates for which this is true. In order to increase the accuracy of the abstraction, usually the negations of all predicates are added to the precision before the abstraction is computed. This method queries the SMT solver $|\pi|$ times with relatively small input problems, and the computed result is always a conjunction of predicates from π (which can be stored very easily).

2.2.2 Boolean Predicate Abstraction

If the precision achieved with Cartesian predicate abstraction is not enough, *Boolean predicate abstraction* as defined in [LNO06] can be used. Therefor only one query to an SMT solver is needed, but the input formula is larger and instead of a mere satisfiability check, all satisfying assignments of the formula need to be enumerated. For each predicate $p_i \in \pi$, a propositional variable v_i is added. The SMT solver is asked for all assignments to these variables that let the formula $\varphi \wedge \bigwedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$ be true. For each such assignment $m : \{v_1, \ldots, v_n\} \to \mathbb{B}$ the conjunction over all predicates whose propositional variable is set to *true* in the assignment is constructed, i.e., $\bigwedge \{p_i \in \pi \mid m(v_i)\}$. The result, the Boolean abstraction $\varphi^{\pi}_{\mathbb{B}}$ for φ, is the disjunction of all these formulas. This abstraction is more precise, as it contains not only those predicates that are unconditionally entailed by φ, but also conjunctions of predicates. However, computing the abstraction is much more expensive and the result can be much larger, as the number of satisfying assignments can be exponential in the size of π.

2.3 Paths as Formulas

Computing an abstraction of an abstract state happens at one program location at a time, using only the abstraction of the predecessor state and the operation attached to the edge between them. When it is necessary to view bigger parts of the program as one formula, a problem arises. Assignments cannot be represented directly in a Boolean formula. This problem can be eliminated by using ideas from the Static Single Assignment (SSA) form [CFR$^+$91] for programs [BHJM07]. Each variable is assigned only once. Whenever a new assignment $x := e'$ to a variable x occurs, a new variable x' is introduced, and all further references to the original variable x in the rest of the program are replaced by references to x'. This can be implemented by adding an index to each variable of the program, which is increased by one on every assignment. In SSA form it is possible to represent the assignment by the formula $x' = e'$. It is not necessary to transform the program into an SSA form before analyzing it, instead the necessary adjustments can be made when a formula is constructed from a part of the program. A *path formula* φ is a formula representing one or more (alternative) paths in the CFA such that the formula is unsatisfiable if and only if all represented paths are infeasible. Given such a path formula φ, the function $\mathit{index}_\varphi : X \to \mathbb{N}$ returns for any variable $v \in X$ the maximum of all indices of occurrences of v in φ, or zero if there is no such occurrence, i.e., $\mathit{index}_\varphi(v) = \max(\{0\} \cup \{k \mid v_k \text{ occurs in } \varphi\})$. A path formula can be constructed iteratively for a finite sequence $\langle (l_1, op_1, l_2), \dots, (l_n, op_n, l_{n+1})\rangle$ of CFA edges. The initial formula φ_0 is *true*. For each $i \in \{1, \dots, n\}$, the path formula φ_i is constructed as follows: If op_i is an assume operation of the form $\mathit{assume}(p)$, the path formula φ_i is set to $\varphi_{i-1} \wedge p'$, where p' is created from p by replacing any free variable v in p by v_j with $j = \mathit{index}_{\varphi_{i-1}}(v)$. If op_i is an assignment operation of the form $x := e$, the path formula φ_i is set to $\varphi_{i-1} \wedge (x_j = e')$, where $j = \mathit{index}_{\varphi_{i-1}}(x) + 1$ and e' is created from e by replacing any free variable v in e by v_k with $k = \mathit{index}_{\varphi_{i-1}}(v)$. Therefore, the path formula for a path without branches is always a conjunction of the formulas representing single operations, only with indices added to all occurrences of free variables. Given two path formulas φ_1 and φ_2, the path formula for the disjunction of the two paths is $\varphi = (\varphi_1 \wedge \psi_1) \vee (\varphi_2 \wedge \psi_2)$, where ψ_1 and ψ_2 are two correction terms equalizing the index of each variable. Both ψ_1 and ψ_2 are conjunctions. For any variable v that occurs in φ_1 or in φ_2, the term $v_i = v_j$ with $i = \mathit{index}_{\varphi_1}(v)$ and $j = \mathit{index}_{\varphi_2}(v)$ is created. If $i > j$, this term is added to ψ_2 and if $i < j$ the term is added to ψ_1. Otherwise this term is not needed.

2.4 Counterexample-Guided Abstraction Refinement and Lazy Abstraction

In the previous section, the precision was assumed to be given externally. The predicates for the precision have to be chosen wisely, as they are the key for a good performance and a successful analysis. Choosing too few predicates would result in too imprecise abstractions and potentially failing to prove that a program is safe, while too many predicates would result in overly expensive abstraction computations. With *counterexample-guided abstraction refinement* (CEGAR) [CGJ$^+$03] a technique called *lazy abstraction* [HJMS02] is possible which automatically finds the needed predicates and uses only these. Analysis starts with the empty set as the initial precision. This will result in all abstractions being *true* and the error location to be reachable (if it is syntactically reachable in the CFA). When a path σ to the error location (a *counterexample*) is encountered, this path is checked for feasibility by checking the satisfiability of $\mathsf{SP}_\sigma(true)$. If the path is indeed feasible, the program is unsafe and analysis terminates. If the path is not feasible, the counterexample is said to be *spurious*. In this case, the path is analyzed, the predicates necessary to prove this path as infeasible are extracted [HJMM04] and added to the precision, and the analysis is restarted with the new precision. This is done iteratively until either a real counterexample is found or no path to the error location is feasible any more.

The example from Figure 2.1 has been analyzed with lazy predicate abstraction. The resulting abstract reachability tree which contains all reachable abstract states can be seen in Figure 2.3.

2.4.1 Abstract Reachability Tree

In order to analyze the path of a counterexample, the information which abstract state is the predecessor of a given state has to be known. This is done by creating an *abstract reachability tree* (ART) [BHJM07]. This tree stores the generated abstract states together with the program location they belong to, and the predecessor-successor relation. The nodes of this tree are tuples (l, φ) of program locations and abstract states. An edge $((l, \varphi), (l', \varphi'))$ is contained in the tree, if there exists an edge (l, op, l') in the CFA and $\varphi' = \mathsf{SP}^\pi_{op}(\varphi)$. The root of an ART is the node $(l_0, true)$. A node (l, φ) is called *covered* if there is another non-covered node (l', φ') in the ART which belongs to the same program location and whose abstract state entails the covered node's abstract state, i.e., $(l' = l) \wedge (\varphi' \models \varphi)$. The successors of a covered node do not need to be analyzed further, because they are already over-approximated by the successors of the covering node. Thus an ART is called *complete* if every node is either covered or all its possible successors exist as its children in the ART. A complete ART that does not contain a node $(l_E, \cdot)$ for the error location l_E exists only for l_E-safe programs.

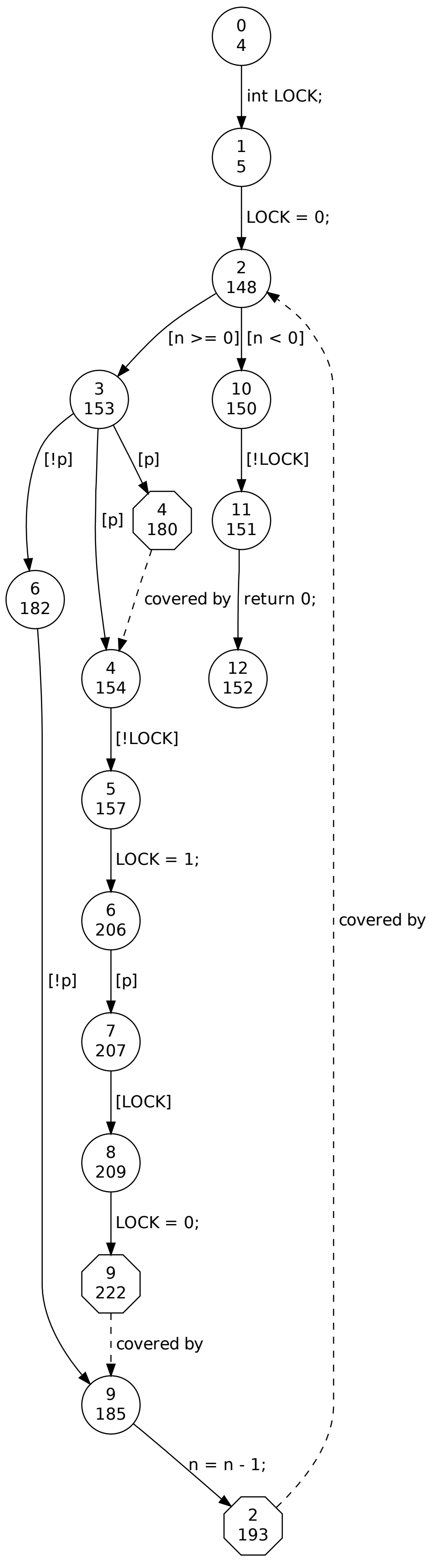


Figure 2.3: ART for `lock-loop.c`
Each node is labeled with two numbers, the first being the number of the CFA node it belongs to and the second a unique identifier assigned to all ART nodes. The ART node identifiers are non-contiguous due to the use of refinement, which results in the deletion of ART nodes and a (partial) restart of the analysis. If the ART node number of a node φ is larger than the ART node number of a node ψ, the node φ has been generated later than node ψ. The edges of the ART are represented only by the solid lines. These are labeled with the operations from the corresponding CFA edges for better orientation. Nodes whose shape is an octagon are covered nodes. Such nodes have an outgoing dashed line that shows which node is responsible for a node to be covered.
Abstract states which are not reachable are not contained in the ART. This can be seen at node $^{10}_{150}$ for example. It has only one successor although the CFA node 10 has two outgoing edges. The analysis has correctly determined that the other path, which leads to the error location, is not feasible, as the edge connecting node 10 and the error location is labeled with $assume(LOCK)$, and $LOCK$ is never equal to $true$ at location 10.
The final precision found through refinement for this example is:

$$\Pi(l) = \begin{cases} \{LOCK, p\} & \text{if } l \in \{4, 6\} \\ \{LOCK\} & \text{if } l \in \{2, 3, 6, 7, 9, 10\} \\ \{p\} & \text{if } l \in \{5\} \\ \emptyset & \text{otherwise} \end{cases}$$

It can be seen that each predicate is only added to the precision where it is needed. For example, the predicate $LOCK$ is not present for location 5 and 8 although it is contained in the precision for all other locations of the loop body because the value of the variable `LOCK` will be overwritten immediately after these two locations anyway.

2.4.2 Generating Predicates via Craig Interpolation

Given an infeasible path σ, it is necessary to decide which predicates are useful to prove that this path is infeasible, and where (on which program locations) they are useful. Craig interpolants [Cra57] can be used to generate the right predicates [HJMM04]. Given two formulas φ^- and φ^+ whose conjunction is unsatisfiable, a *Craig interpolant* of φ^- and φ^+ is a formula ψ which fulfills the following properties:

1. $\varphi^- \Rightarrow \psi$
2. $\psi \wedge \varphi^+$ is unsatisfiable
3. ψ contains only variables which occur in φ^- and φ^+

Therefore, the interpolant ψ is a formula which contains enough information from φ^- to make $\psi \wedge \varphi^+$ unsatisfiable, but is hopefully smaller than φ^-. As it is entailed by φ^-, the unsatisfiability of $\psi \wedge \varphi^+$ can be used to show the unsatisfiability of $\varphi^- \wedge \varphi^+$. Interpolants can be generated from the proof of unsatisfiability of $\varphi^- \wedge \varphi^+$. To find predicates for the infeasible path σ which is represented by its path formula φ, the path is split at a location l. The part of φ that corresponds to the first part is used as φ^-, the other part of φ is used as φ^+. Then the interpolant for these two formulas contains all those statements about the variables of the program, that have to be known at the cut point l in order to show that the remainder of the path is infeasible. Thus all predicates contained in the interpolant are added to the precision at location l. This can be done for every location in the path.

An SMT solver that supports Craig interpolation is likely to offer a method for checking the satisfiability of a conjunctive formula and generating the interpolants for all possible cut points, so that less queries have to be executed and redundant computations can be reduced. This also guarantees that all interpolants are generated from the same proof of unsatisfiability, which is necessary in order to ensure that all interpolants together prove that the path is infeasible. Since interpolation is used for model checking, efficient algorithms have been developed and implemented that provide interpolants for theories like LA+EUF [CGS08] [BZM08].

2.5 Large-Block Encoding

The traditional approach to predicate abstraction, which was explained in the previous sections, can generate large amounts of abstract states, and needs many abstraction computations for this. Thus, a new approach called Large-Block Encoding (LBE) [BCG+09] was proposed, which reduces the number of abstract states by combining CFA edges into blocks. For differentiation, the previous approach will be called Single-Block Encoding (SBE), as it uses blocks that contain only a single CFA edge. LBE's blocks, however, can contain large parts of the program, including several control-flow branches, but excluding loops. This can lead to an exponential reduction in the number of abstract states. The queries given to the SMT solver for abstraction computation can be exponentially larger, but today's solvers employ efficient heuristics for satisfiability checks, so that there is an overall performance increase of one to two orders of magnitude.

The original CFA is transformed into a new "summarized" CFA containing only large blocks by applying three rules:

Rule 0: All outgoing edges from the error location l_E are removed.

Rule 1: Each node $l' \in L \setminus \{l_E\}$ which has only one incoming edge (l, op, l') is removed, together with all its adjacent edges. For each removed outgoing edge (l', op', l'') a new edge $(l, op; op', l'')$ is introduced. This edge connects the predecessor l and the successor l'' and is labeled with the sequential combination of the operations from the two edges it replaces. Note that l and l'' may be equal.

Rule 2: Any two edges (l, op_1, l') and (l, op_2, l') which share the same predecessor and successor are replaced by a single edge $(l, op_1 \| op_2, l')$ which is labeled with the disjunctive combination of the operations from the two replaced edges. For the disjunctive combination, the strongest postcondition operator is defined as $\mathsf{SP}_{op_1 \| op_2}(\varphi) = \mathsf{SP}_{op_1}(\varphi) \vee \mathsf{SP}_{op_2}(\varphi)$.

Rule 0 is applied once, and Rule 1 and Rule 2 are applied repeatedly until a fixpoint is reached. Then each edge of the new CFA is labeled with the operations from a block of the program. It is ensured that such a block does not contain loops, because in a loop there is at least one CFA node which has two incoming edges, and such nodes are never removed from the CFA. A loop that does not contain nested loops nor the error location is now represented by a single CFA node for the loop head with a reflexive edge that is labeled with all operations contained in the loop body.

A summarized CFA can be used for predicate abstraction with the extended strongest postcondition operator, because the error location is reachable in the new CFA if and only if it is reachable in the original CFA. The summarization does not change the semantics of the program [BCG+09]. Due to the increased complexity of the formulas however, it is necessary to use Boolean predicate abstraction. Carte-

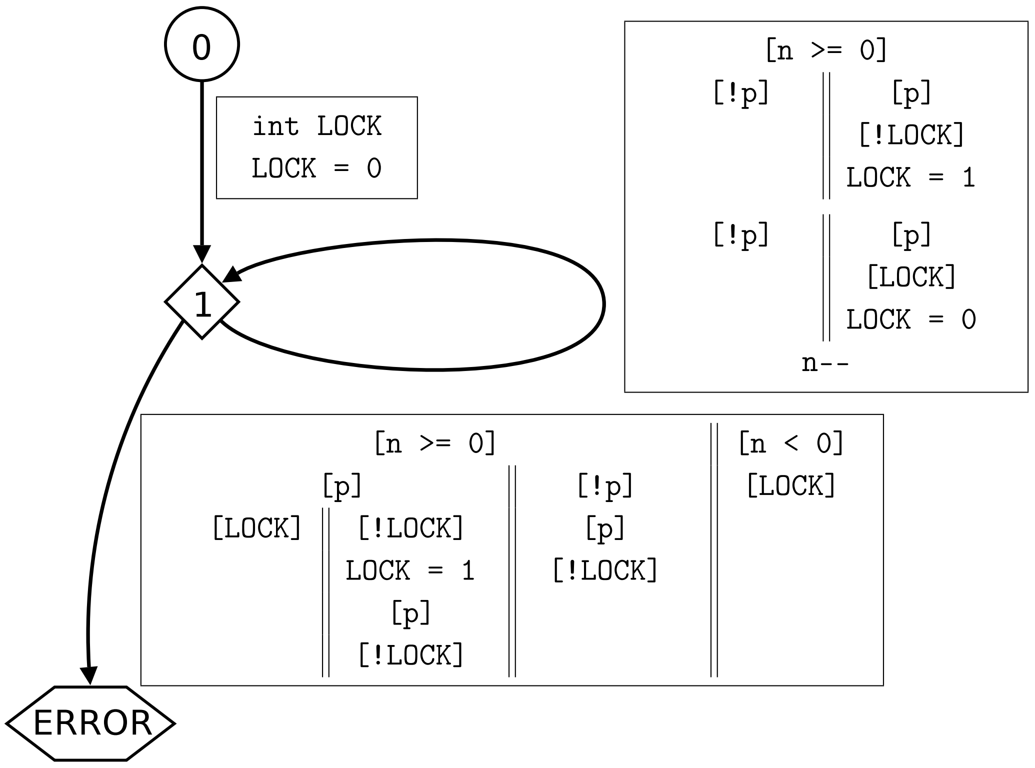
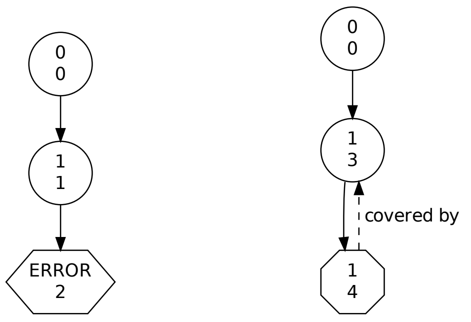


Figure 2.4: Summarized version of the CFA in Figure 2.2 (example `lock-loop.c`) with the labels of the edges shown in the boxes. $assume(p)$ is represented by `[p]`, alternative execution is represented by $\|$ and op_1 ; op_2 is represented by putting op_2 under op_1



(a) ART before first refinement (b) Final ART

$$\Pi(l) = \begin{cases} \{LOCK\} & \text{if } l = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

(c) Final Precision

Figure 2.5: Results produced by LBE for `lock-loop.c`

sian predicate abstraction is not precise enough and does not succeed in verifying reasonable-sized programs.

The effect of the summarization when applied to the CFA from Figure 2.2 can be seen in Figure 2.4. Figure 2.5 shows result of analyzing this example with LBE. First, the analysis starts with an empty precision, so the path to the error location is found and refinement is used. At this point, the ART looks like in Figure 2.5a. Refinement finds the predicate $LOCK$ to be necessary at location 1. Therefore the ART nodes 1 and 2 are removed and analysis is restarted at node 0. With the new precision that can be seen in Figure 2.5c the error location is not reachable anymore. The analysis finishes with the ART from Figure 2.5b without needing any further refinement.

2.6 On-the-fly Large-Block Encoding

Compared with Single-Block Encoding, Large-Block Encoding greatly increases the performance of the analysis, but has the disadvantage that it needs a pre-processed CFA. This is inconvenient if the analysis uses several abstract domains like predicate abstraction together with the explicit tracking of some variables, because the other domains would need to be adapted to summarized CFAs as well. It is also not possible to dynamically adjust the size of the blocks, which could be used to maintain the best trade-off between the number of blocks (respectively the number of abstraction computations) and the size of the blocks (respectively the complexity of the abstraction computations). Thus, a new approach called *On-the-fly Large-Block Encoding* which works with an unmodified CFA was developed by Dirk Beyer and M. Erkan Keremoglu, but has not been published yet. It is similar to LBE as abstractions are computed only at program locations corresponding to loop heads and the error location (such locations are called *abstraction locations*). For any CFA, there is a bijective mapping between its abstraction locations and the locations the CFA would have after summarization, as the summarization removes exactly the non-abstraction locations from the CFA.

With On-the-fly LBE, each abstract state consists of a pair (ψ, φ) where ψ is the result of an abstraction computation (an *abstraction formula*), and φ is a path formula as defined in Section 2.3. Given an edge (l, op, l') and an abstract state (ψ, φ), the successor of this state is defined as follows: If l' is an abstraction location, φ' is set to $\mathsf{SP}_{op}(\varphi)$ and $\psi_{\varphi'}$ is created from ψ by replacing each variable v that occurs in ψ with v_i where $i = index_{\varphi'}(v)$. Then the successor is $(\psi', true)$ with ψ' being created from the abstraction of $\varphi' \wedge \psi_{\varphi'}$ by removing the indices attached to the variables. In other words, the abstraction of the conjunction of φ, the formula representing the current operation op and ψ (with the right indices added to all variables) is computed, with the indices removed from all variables afterwards. If l' is not an abstraction location, the successor is (ψ, φ'), where φ' is constructed from the path formula φ and the operation op as described in Section 2.3. If control flow meets at a non-abstraction location and there are two abstract states (ψ_1, φ_1) and (ψ_2, φ_2), these two abstract states are merged into a new abstract state (ψ', φ') if ψ_1 and ψ_2 were computed at the same program location and are equal. The abstraction formula of the merged node is $\psi' = \psi_1 = \psi_2$ and the path formula φ' is also constructed from φ_1 and φ_2 as described in Section 2.3. This ensures that for any abstract state that does not belong to an abstraction location, the path formula represents all possible loop-free paths to the last abstraction location.

When using the same precision, the result of On-the-fly LBE is equal to that of LBE with a summarized CFA, as there is a bijective function mapping the abstract states of the latter to those abstract states of the former that belong to an abstraction location. This function maps an abstract state ψ produced by LBE with pre-processing to an abstract state (ψ, true) of On-the-fly LBE. Figure 2.6 shows the resulting ART for analyzing `lock-loop.c` with On-the-fly LBE and demonstrates this point.

2.6.1 Refinement strategy

Predicate abstraction with on-the-fly creation of large blocks can be used with ART-based refinement. After a path to the error location is found, the formula representing this path has to be created. This is different from predicate abstraction without Large-Block Encoding since abstract states on the path may have been merged. Therefore there is no ART anymore, but instead a directed acyclic graph (DAG) that contains the same nodes and is constructed in the same way, except that two nodes can be merged with the resulting node having all parents and children of the two nodes. As there is not much difference to a tree (e.g. there still is a single root node) and out of tradition, this data structure will still be called ART. With Large-Block Encoding, some properties hold that make the ART even more similar to a tree and which help to construct the formula representing the path(s) to the error location. An ART node whose abstract state was produced by an abstraction computation is called abstraction node. For any node in the graph, let the nearest abstraction node be the node itself, if it is an abstraction node, and the nearest abstraction node among the ancestors of the node otherwise. Then, the definition of the On-the-fly Large-Block Encoding ensures that only such pairs of nodes are merged, where both nodes belong to a non-abstraction location and both have the same nearest abstraction node. This means that those edges that violate the tree property of the ART can connect only nodes which would have been close in the graph anyway. So on a higher level, the ART can still be seen as a tree with several parts that are each a DAG. These DAG parts do not

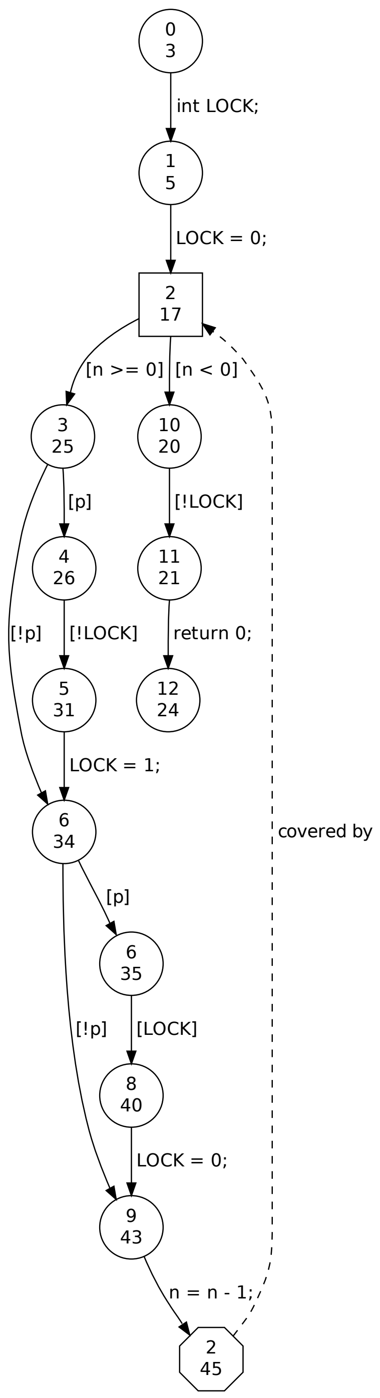


Figure 2.6: ART produced by On-the-fly LBE for `lock-loop.c`
In this ART, nodes with a square or an octagon as their shape represent nodes which were produced by computing an abstraction. Octagon nodes are nodes which are covered, like in Figure 2.3. With LBE, only abstraction nodes can be covered, as the coverage relation is based only on the abstraction formula and therefore needs to be computed only for abstraction nodes.
The three nodes $\frac{0}{3}$, $\frac{2}{17}$ and $\frac{2}{45}$ correspond exactly to the three nodes of the final ART produced by LBE that can be seen in Figure 2.5b. The number of abstraction computations and refinement steps as well as the final precisions are also equal for LBE with a summarized CFA and On-the-fly LBE. The ART of the latter just contains additional non-abstraction nodes, but these nodes are very inexpensive to create, as only a syntactical transformation from code to predicates is necessary.
Therefore, On-the-fly LBE is more similar to LBE with pre-processing, although at first glance its ART looks more similar to the ART produced by SBE. However, some differences do occur. With SBE, the two ART nodes that belong to program location 6 are not merged, whereas with LBE they are merged, producing node $\frac{6}{34}$ that has two parents. The same is true for the nodes at location 9. In a larger program this could result in the LBE ART being much smaller.

contain any abstraction nodes, as such nodes are never merged. Thus, for every node all paths from this node to the root of the ART contain exactly the same abstraction nodes in the same order, and only the non-abstraction nodes may vary. This can be used to construct the path formula which represents all paths from the node on the error location to the root node as a conjunction of the path formulas for each part between two abstraction nodes. Only the SSA indices for the variables occurring in the formula need to be adjusted.

If the created path formula is unsatisfiable (which means that all represented paths are infeasible), the cut points for which the interpolants are generated correspond exactly to the abstraction nodes on the path(s) to the error location, as the path formula is a conjunction of the parts between the abstraction nodes. Thus the predicates that are extracted from the interpolants get added to the precision Π at the right locations so that they will be used in the next iteration of the abstraction-refinement loop to compute more precise abstractions at the abstraction locations.

2.7 Configurable Program Analysis

Configurable Program Analysis [BHT07] is a framework for software verification that can be used as a precise model checker as well as as an efficient lattice-based program analyzer. It defines an algorithm (shown as Algorithm 1) that uses an abstract interpreter (called *configurable program analysis* (CPA)) providing the domain of the analysis and some operators. The result of the CPA algorithm is the *reached set* R which contains all reachable abstract states of the program. CPAs can and have been defined for a lot of known software verification domains, like predicate abstraction, shape-based heap analysis and others. Several CPAs can be combined and executed simultaneously by using the Composite pattern known from software engineering [GHJV94]. A CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge and a termination check stop.

1. The *abstract domain* $D = (\mathcal{C}, \mathcal{E}, [\![\cdot]\!])$ is defined by a set $\mathcal{C}$ of concrete states, a semi-lattice $\mathcal{E}$ and a concretization function $[\![\cdot]\!]$. The lattice $\mathcal{E} = (E, \top, \sqsubseteq, \sqcup)$ consists of a set E of lattice elements (the abstract states of the analysis), a top element $\top \in E$, a preorder $\sqsubseteq \subseteq E \times E$ and a join operator $\sqcup : E \times E \rightarrow E$. The concretization function $[\![\cdot]\!] : E \rightarrow 2^{\mathcal{C}}$ assigns to an abstract state the set of concrete states it represents.

Algorithm 1 $CPA(\mathbb{D}, e_0)$

```
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 of elements of $E$,
       a set waitlist of elements of $E$
 1: waitlist := $\{e_0\}$
 2: reached := $\{e_0\}$
 3: while waitlist $\neq \emptyset$ do
 4:   choose $e$ from waitlist
 5:   waitlist := waitlist $\setminus \{e\}$
 6:   for each $e'$ with $e \rightsquigarrow e'$ do
 7:     for each $e'' \in$ reached do
 8:       // combine with existing abstract state
 9:       $e_{new}$ := merge($e', e''$)
10:       if $e_{new} \neq e''$ then
11:         waitlist := $(\text{waitlist} \cup \{e_{new}\}) \setminus \{e''\}$
12:         reached := $(\text{reached} \cup \{e_{new}\}) \setminus \{e''\}$
13:     if $\neg$ stop($e'$, reached) then
14:       waitlist := waitlist $\cup \{e'\}$
15:       reached := reached $\cup \{e'\}$
16: return reached
```

2. The *transfer relation* $\rightsquigarrow \ \sqsubseteq E \times G \times E$ defines which elements are the successors of an element, given an edge from the CFA.
3. The *merge operator* $\mathsf{merge} : E \times E \rightarrow E$ may combine the information of two abstract states. It can weaken the second parameter using the information of the first parameter, so that the resulting element represents more concrete states. The operator which always returns the second parameter (and thus performs no weakening and no loss of precision) is called $\mathsf{merge}^{\mathsf{sep}}$. The operator which for two elements $e_1, e_2 \in E$ always returns $e_1 \sqcup e_2$ (returning an element which represents at least the union of the sets of concrete states represented by e_1 and e_2 respectively) is called $\mathsf{merge}^{\mathsf{join}}$. While this operator reduces the precision of the analysis, it also increases the performance, as only the successors of the newly created element need to be constructed, and not the successors of e_1 and e_2. Using $\mathsf{merge}^{\mathsf{sep}}$, the precision of a model checker is used, while with $\mathsf{merge}^{\mathsf{join}}$, the performance of a lattice-based program analyzer can be achieved.
4. The *stop operator* $\mathsf{stop} : E \times 2^E \rightarrow \mathbb{B}$ determines whether a given abstract state can be considered covered by the set of abstract states already reached. If so, the analysis does not need to consider the successors of this abstract state. The

most commonly used implementation of stop is the operator $\mathsf{stop}^{\text{sep}}$ which checks if there is any reached node which represents at least all concrete states of e, i.e., $\mathsf{stop}^{\text{sep}}(e, R) = (\exists e' \in R : e \sqsubseteq e')$.

2.7.1 State-Space Traversal Methods

The CPA framework does not define the order in which the reachable abstract elements are constructed. However, it is necessary to choose the right strategy (that defines the order in which elements are taken from the waitlist) to gain good performance. One possibility for this is to use a stack for the waitlist, so that the last added element is the next one to be used. This constructs the abstract reachability tree in the same order a depth-first iteration over the tree would enumerate the elements in. This strategy has the advantage that one path gets fully analyzed before other paths are tried, which finds an eventual error location at the end of a path quite fast. If this error location is indeed reachable, the analysis can terminate as it proved the program to be unsafe. The opposite strategy is to use a queue for the waitlist, which would result in a breadth-first-like iteration order. This is generally slower because the beginnings of all paths are analyzed before an eventual error location at the end of a path is found and analyzed. However, for configurations using other merge operators than $\mathsf{merge}^{\text{sep}}$ (which never merges), depth-first iteration might not be optimal. When two paths are merged and the first path is weakened, the rest of the first path has to be re-analyzed. So it would have been better to not analyze the first path in its full length but instead to wait with analyzing the first path until the second path has been merged into it. However, it is not possible to know in advance if a given abstract state will be merged with another one. So a breadth-first iteration might be faster assuming that when a merge occurs, both paths are often of similar length. For example, this might be the case when the control flow meets after two similarly long branches. But for analysis which explicitly model the program counter variable and never merge two abstract states which belong to different locations, a better strategy is to use an iteration order similar to topological sort. In this case, every program location gets a number assigned which is higher than the number of all its predecessor locations (ignoring backwards edges). For the waitlist, a priority queue is used which always returns one of those abstract states that belong to the location with the lowest number among all waitlist states' locations. This leads to the behavior that the successor location of a control-flow meet point is analyzed only after the paths leading to that location have been analyzed (and their abstract states have been merged).

3 Adjustable Large-Block Encoding

On-the-fly Large-Block Encoding has the advantage of working on an unmodified CFA and still providing the benefits of LBE, but it is not flexible enough to allow different block sizes to be used. *Adjustable Large-Block Encoding* is an analysis that is based on On-the-fly LBE, but the block size it uses is configurable. By modifying one parameter, its behavior can not only be switched between Single-Block Encoding and Large-Block Encoding, but also changed to any of numerous other configurations with block sizes between SBE and LBE as well as block sizes larger than LBE. With On-the-fly LBE, the decision whether an abstraction should be computed for a new abstract state depends only on the current program location. Abstractions are always computed at loop heads and at the error location. For Adjustable LBE, this decision is no longer hard-wired, but instead made by a new operator called *abstraction operator* abs, which takes an abstract state and a CFA edge as input. It returns *true* if an abstraction should be computed and $false$ otherwise. Only at the error location an abstraction computation is still enforced regardless of the decision made by the new operator, because this is needed for the analysis to determine if an error location is unreachable. The actual generation of the successor (that is either the computation of the abstraction or the creation of the path formula) is not changed compared to On-the-fly LBE.

The great flexibility of Adjustable Large-Block Encoding results from the possibility to choose the abs operator freely. Two particular choices for abs are $\mathsf{abs}^{\mathsf{SBE}}$, which specifies to always compute an abstraction, and $\mathsf{abs}^{\mathsf{LBE}}$, which specifies to compute an abstraction if the successor location of the current CFA edge is an abstraction location. Thus the analysis can be configured to behave exactly like SBE or LBE. Another possible choice for abs would be to compute an abstraction if the length of the longest path represented by the path formula φ of the abstract state exceeds a certain threshold. This operator is named abs_k for any threshold $k \in \mathbb{N} \setminus \{0\}$. But the decision made by abs does not necessarily have to be based only on statically available information. One implementation could for example measure the free memory of the system and compute abstractions if the path formulas need too much memory.

Another could benchmark the time needed to compute the abstractions and adjust the block size so that a single computation does not take too much time. The decision whether an abstraction should be made could also depend on the precision Π, e.g. such that abstractions are computed when the precision contains predicates for the current program location.

It should be noted that the analysis will not terminate if there is an infinite walk through the CFA for which the chosen implementation of abs will never specify to compute an abstraction. This happens for example if the abs operator always returns $false$ (so that abstractions are never computed) and the analyzed program contains a loop. Therefore it is recommended to use an implementation of abs that has an upper bound on the size of the blocks it produces. For $\mathsf{abs}^{\mathsf{LBE}}$ this is guaranteed as the size of the CFA is finite, and thus any longer path will contain loops.

Adjustable Large-Block Encoding allows abstract states that belong to the same program location to be merged under some conditions. Firstly, an abstract state that was generated by computing an abstraction may never be merged with any other node. Secondly, two abstract states with different abstractions may never be merged. These two conditions are required to prevent a great loss of precision (the analysis would become similar to a lattice-based program analyzer otherwise). Thirdly, abstract states may only be merged if their abstractions were computed at the same program location. This is necessary to ensure that all paths in the ART from one ART node to the root contain exactly the same abstractions, as explained in Section 2.6.1. With these conditions, the ART produced by Adjustable LBE for any program is similar to the one produced by On-the-fly LBE for the same program, the only difference being the size of the parts between the abstraction nodes. Because of this, the same refinement strategy can be used.

If the block size produced by the chosen abstraction operator is large enough that the whole program with each loop unrolled a few times fits into one block (and is therefore represented by a single path formula), the analysis becomes similar to Bounded Model Checking (BMC) [BCCZ99]. However, there are still some differences as BMC does not use abstractions at all and does not try to give a sound verification result.

For SBE and LBE it is easy to decide which abstraction mechanism is used for the analysis. Cartesian abstraction is faster than Boolean abstraction for SBE, but not powerful enough for LBE [BCG+09]. With Adjustable Large-Block Encoding this decision is still important, but more difficult to make, as there are many more possible configurations.

3.1 CPA for Adjustable Large-Block Encoding

The formal definition of Adjustable Large-Block Encoding is given as a CPA. Predicate abstraction has already been expressed as a CPA [BHT07], proving its applicability. An advantage is that CPAs can be easily implemented due to an available ready-to-use implementation of a framework for CPAs, requiring only the operators of the analysis to be implemented.

The CPA for Adjustable Large-Block Encoding $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ is based on the original CPA for Predicated Abstraction [BHT07] and the concepts of LBE [BCG+09] and On-the-fly LBE (Section 2.6). Given a set X of program variables and a control-flow automaton $A = (L, G, l_0)$ with a set L of program locations, a set G of edges as well as an initial node l_0, let $l_E \in L$ be the error location of the program, let $\mathcal{P}$ be the set of quantifier-free predicates over variables from X, let $\mathcal{F}$ be the set of arbitrary quantifier-free formulas over variables from X and let $\Pi : L \to 2^{\mathcal{P}}$ be a precision. The components of $\mathbb{D}$ are defined as:

1. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ is a tuple of a set C of concrete states, a semi-lattice $\mathcal{E}$ of abstract states and a concretization function $[\![\cdot]\!] : \mathcal{E} \to C$. The semi-lattice $\mathcal{E} = (E, \top, \sqsubseteq, \sqcup)$ is defined as:
   a) The lattice elements $e \in E$ (the abstract states) are tuples $(l, \psi, l^\psi, \varphi) \in (L \cup \{l_\top\} \times \mathcal{F} \times L \cup \{l_\top\} \times \mathcal{F})$ where l is the explicitly modeled program counter variable, ψ is a formula over predicates from $\mathcal{P}$ (called the abstraction formula, because it is always the result of an abstraction computation), l^ψ is the location at which ψ was computed and φ is a path formula representing some or all of the paths from l^ψ to l. The formula l^ψ is also called the *abstraction location.* An *abstraction element* is an element which was generated by computing an abstraction. Such elements always have $l = l^\psi$ and $\varphi = \mathit{true}$. An abstract state is reachable, if both ψ and φ are satisfiable.
   b) The top element $\top$ is $(l_\top, \mathit{true}, l_\top, \mathit{true})$ represents a state without any information about the analyzed program.
   c) The partial order $\sqsubseteq \subseteq E \times E$ is defined such that for any two elements $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2)$ from E the following holds:

$$\begin{aligned} e_1 \sqsubseteq e_2 \iff & (e_2 = \top) \\ & \vee \big((l_1 = l_2 = l^{\psi_1} = l^{\psi_2}) \wedge (\psi_1 \implies \psi_2)\big) \\ & \vee \big((l_1 = l_2) \wedge (l^{\psi_1} = l^{\psi_2}) \wedge (\psi_1 = \psi_2) \wedge (\varphi_1 \implies \varphi_2)\big) \end{aligned}$$

This means that for two abstraction elements, the partial order is based on the relation of the abstraction formulas, whereas for two non-abstraction elements with the same abstraction formula, it is based on the relation of the path formulas.

d) The join operator $\sqcup : E \times E \to E$ always returns the smallest (as defined by the partial order) element that is larger than both elements given as parameters.

2. The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ contains all tuples (e, g, e') with $g = (l, op, l')$, $e = (l, \psi, l^{\psi}, \varphi)$ and $e' = (l', \psi', l^{\psi'}, \varphi')$ for which, given a precision $\pi = \Pi(l')$, the following holds:

$$\begin{cases} \left(\psi' = \mathsf{SP}^{\pi}_{op}(\varphi \wedge \psi)\right) \wedge (\varphi' = \mathit{true}) \wedge (l^{\psi'} = l') & \text{if } \mathsf{abs}(e, g) \vee (l' = l_E) \\ (\varphi' = \mathsf{SP}_{op}(\varphi)) \wedge (\psi' = \psi) \wedge (l^{\psi'} = l^{\psi}) & \text{otherwise} \end{cases}$$

This transfer relation is adjustable by choosing an abstraction $(\cdot)^{\pi}$ (i.e., whether to use Cartesian abstraction or Boolean abstraction) and an abstraction operator $\mathsf{abs} : E \times G \to \mathbb{B}$. The choice of abs will determine the size of the blocks. Two particular possibilities are $\mathsf{abs}^{\mathsf{SBE}}$ which returns always *true* and $\mathsf{abs}^{\mathsf{LBE}}$ which returns *true* if the successor location of the given edge is the head location of a loop. It is suggested that an abs operator is chosen that will eventually return true for every path through the CFA, otherwise the analysis would not terminate if the program contains loops.

3. The merge operator $\mathsf{merge} : E \times E \to E$ is defined for two abstract elements $e_1 = (l_1, \psi_1, l^{\psi_1}, \varphi_1)$ and $e_2 = (l_2, \psi_2, l^{\psi_2}, \varphi_2)$ from E as

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

This is equivalent to using $\mathsf{merge}^{\mathrm{join}}$ if both elements belong to the same location and their abstraction formulas are equal and were computed at the same location, and $\mathsf{merge}^{\mathrm{sep}}$ (i.e., no merging) otherwise.

4. For the stop operator $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ the $\mathsf{stop}^{\mathrm{sep}}$ operator is used, i.e.,

$$\forall e \in E, R \subseteq E : \mathsf{stop}(e, R) = (\exists e' \in R : e \sqsubseteq e')$$

3.2 Example

In order to give an example, the program `lock-loop.c` from Figure 2.1 was analyzed with Adjustable LBE. For the abstraction operator, the conjunction of $\mathsf{abs}^{\mathsf{LBE}}$ and abs_{15} was used. This operator returns true if the current location is an abstraction location (that is a loop head) and the length of the longest path represented by the path formula of the current abstract state is at least 15. The length of one iteration of the loop is 8 edges as can be seen in Figure 2.2. Therefore, in this example, an abstraction is made exactly after two iterations.

The resulting ART can be found in Figure 3.1. The node shapes are the same as in Figure 2.6: All non-circular nodes are abstraction nodes, and nodes with an octagon shape are covered. The first number in each node is the program location this node belongs to, and the second number is a unique identifier. The solid lines are the ART edges and are labeled with the operations from the corresponding CFA edges.

This ART was produced in the following way: Firstly, all locations of the program starting a location 0 were analyzed and abstract states were generated for each of them (among them the nodes $\frac{0}{4}$ and $\frac{1}{5}$). Then the successor of the abstract state at location 9 was analyzed, leading to a second node at location 2. These two abstract states were merged because they both had the same abstraction formula and the same abstraction location. The resulting abstract state is represented by node $\frac{2}{37}$. The same occurred at all other locations, producing the nodes $\frac{10}{43}$ to $\frac{9}{81}$. During the generation of the successor of the latter, the abstraction operator returned *true* as location 2 is an abstraction location and the length of the longest path represented by the path formula was 18. So the abstraction node $\frac{2}{92}$ was computed and the rest of the loop was analyzed again. The new nodes were not merged with the existing ones for the same program location because their abstraction and their abstraction location were different. Node $\frac{2}{121}$ also was not merged with node $\frac{2}{92}$ as abstraction nodes like the latter never are. Similarly to before, the nodes generated during the fourth iteration of the loop were merged with the nodes of the third iteration, producing the nodes $\frac{10}{127}$ to $\frac{9}{165}$. The next time an abstract state was produced at location 2, again a new abstraction was computed. As there was already an abstraction node at the same location which covered the new node, no successors for this node were produced. At this point, there were no further abstract states in the waitlist, so the analysis terminated.

Compared to LBE, the same number of abstraction computations were needed, but the input formulas had roughly double the size.

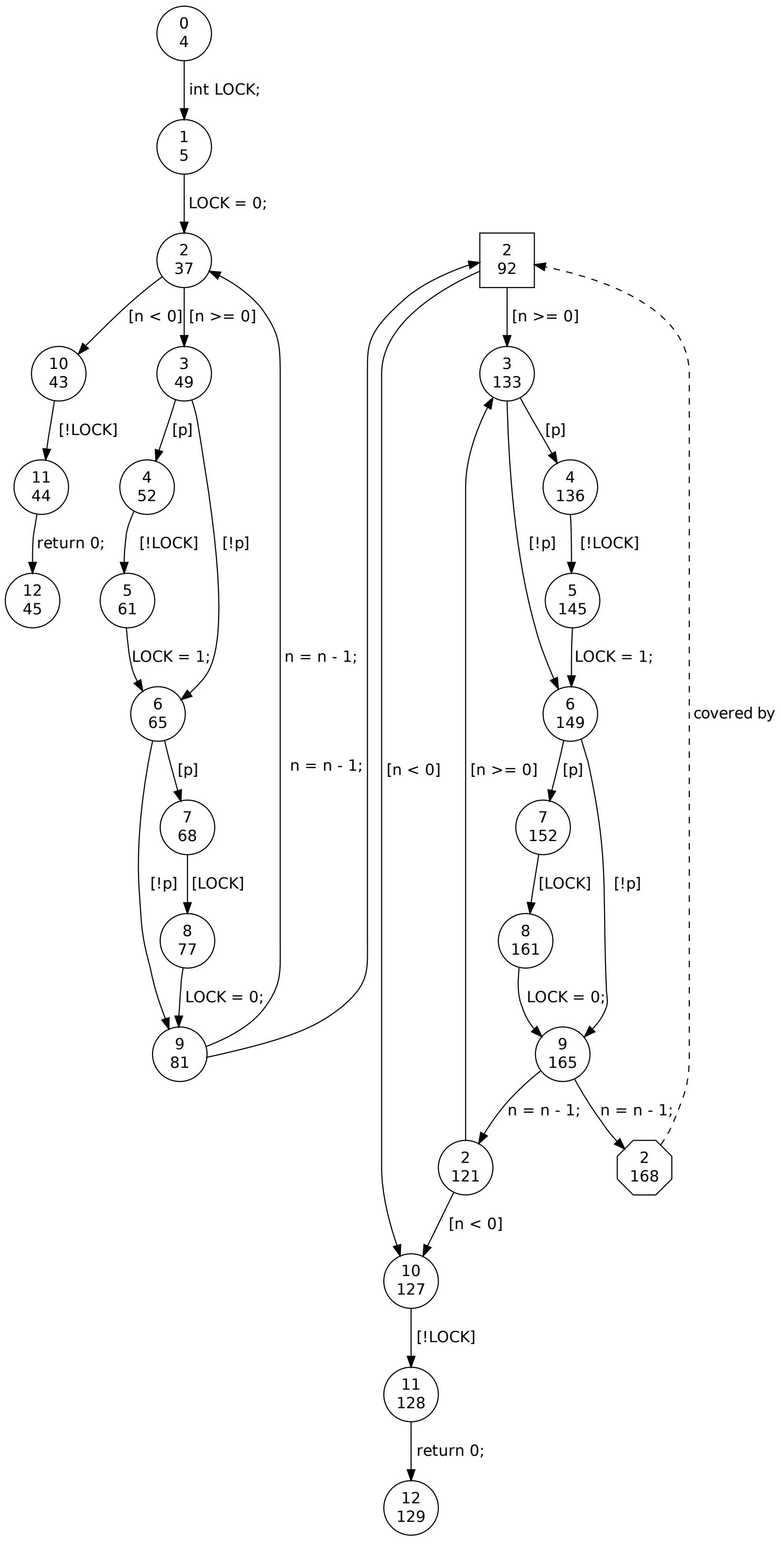


Figure 3.1: ART produced by Adjustable LBE for `lock-loop.c`

4 Implementation

The CPA for Adjustable Large-Block Encoding was implemented as a component of CPAchecker[1] [BK09]. This tool for static analysis is the proposed successor of the widely used model checker Blast [BHJM07]. It is a framework specifically designed to allow CPAs to be implemented and used easily, by providing everything else that is needed. It features a C parser, a component to create a CFA from a parsed source file, an implementation of the CPA algorithm with precision adjustment and an algorithm for counterexample-guided abstraction refinement. CPAs for predicate abstraction providing the ability to use SBE as well as LBE are also contained. These were the implementations used for the benchmarks in [BCG$^+$09]. A CPA for the on-the-fly encoding of the blocks that works on an unmodified CFA was implemented by Dirk Beyer and M. Erkan Keremoglu, but this work has not yet been published. This CPA was extended in order to provide the flexibility needed to run other configurations as well. While theoretically easy, this was more difficult than expected, because the existing code relied on some assumptions that were true for LBE, but no longer for adjustable LBE. These assumptions had to be identified and the code had to be changed appropriately.

The implementation supports interprocedural analysis, handling function calls and managing scoped variables. Similarly to the previous implementation of LBE, additional abstractions are computed after function call and function return edges if $\mathsf{abs}^{\mathsf{LBE}}$ is used. However, recursive functions are not supported by CPAchecker.

An optimization that is already contained in the previous CPA is to not restart the analysis after each refinement step by removing every element from the reached set, but instead to remove only those elements belonging to the infeasible error path. This saves the computation cost of re-generating all other elements.

Another optimization was implemented together with Adjustable LBE. Previously, the CPA stored the predicate map Π as global information, so that whenever an abstraction was computed at a node l, the same predicates $\Pi(l)$ were produced (as long

[1] Available at `http://cpachecker.sosy-lab.org`

as Π had not been modified by refinement). This is not optimal because nodes may occur in multiple paths, which could need different predicates to be proven infeasible. When this happens, not all predicates from $\Pi(l)$ are necessary, but nevertheless increase the computation cost for the abstraction. The situation can be improved by using different precisions for different paths. This was implemented by using the framework for dynamic precision adjustment for CPAs [BHT08]. Each abstract element gets a precision attached to it, which is used for abstraction. A newly generated element inherits the precision from its predecessor. During the refinement step, the new predicates can then be added not only specific to the location where they are needed, but also specific to the path, by only modifying the precisions of the elements along the path. The identity function as the precision adjustment function defined by this extension was used, because modifying the precision in-between two refinement steps is not necessary.

5 Evaluation

Previously only the configurations for SBE and LBE with both Cartesian and Boolean predicate abstraction have been used and evaluated [BCG$^+$09]. The results show that two of these configurations are not useful: LBE with Cartesian abstraction as it failed to solve any tested example because of being to imprecise, and SBE with Boolean abstraction as it is prohibitively slow and not able to solve more examples than SBE with Cartesian abstraction. Of the remaining two configurations, LBE with Boolean abstraction succeeds in all examples and is always faster than SBE with Cartesian abstraction.

As the results showed that the smallest possible block size (and resulting from it, the smallest possible formula size) is not the best one, the question arises what the optimum is. It is reasonable to expect that, when starting with a block size of one, analysis becomes faster when the size is increased, but eventually only until a turning point is reached. For even larger block sizes, analysis would get slower or would not succeed, e.g. because of memory problems due to the increased formula sizes.

A second question was at which block size the switch from using Cartesian abstraction to Boolean abstraction needs to be made, because the analysis would become too imprecise otherwise. If this threshold is sufficiently high, it might be advisable to use a smaller block size which allows using the faster Cartesian abstraction instead of a larger block size with the costly Boolean abstraction.

Another performance tuning possibility is the strategy the CPA algorithm uses to take elements from the waitlist. As LBE uses $\mathsf{merge}^{\mathrm{join}}$ in most locations and re-analyzing an already explored path is costly (because all abstractions on that path have to be recomputed if the abstract state has been weakened), it is advisable to use the topological-sort-like iteration order. For SBE however, the default depth-first-like order has the best performance as explained above. So similarly to the choice of the predicate abstraction, there has to be a threshold beyond that it is better to use topsort instead of DFS.

The configurations evaluated in this work can be divided into two categories. The first one uses an abstraction operator that is implied by $\mathsf{abs}^{\mathsf{LBE}}$, i.e., the operator returns *true* at least if $\mathsf{abs}^{\mathsf{LBE}}$ would return *true*, but potentially more often. This leads to block sizes which are at most as large as with LBE. Note that $\mathsf{abs}^{\mathsf{SBE}}$ is an example of this case. The second category contains all other configurations. These might produce block sizes that are much larger than with LBE.

All benchmarks were run on a virtual machine under VMware on a server. The machine was assigned a quad-core CPU with up to 2800 MHz and 4 GB of RAM. The operating system was the 64bit version of Ubuntu 9.10, using Linux 2.6.31 and OpenJDK 1.6. CPAchecker from the subversion repository in Revision 1213 with MathSAT 4.28 [BCF+08] as the SMT solver was used. The Java VM was restricted to 4 GB of memory and a maximum execution time of 1800 seconds per program.

The test cases are similar to those used in [BCG+09]. They consist of three groups. The first one are the `test_locks_*` examples that were artificially created to produce an ART which is exponential in size when analyzed with SBE. In these examples several nested locks are acquired and released in a loop, similar to the `lock-loop.c` example from Figure 2.1. The number in the name gives the number of locks in the program. The second group are several parts of drivers from the Windows NT kernel. These examples are named `cdaudio`, `diskperf`, `floppy` and `kbfiltr` according to the driver they are from. The last group are the `s3_*` examples, which were taken from the SSH server (`s3_srvr_*`) and client (`s3_clnt_*`). The code contains a simplified version of the state machine handling the communication according to the SSH protocol. This is basically a large loop with a lot of branches, but no function calls. Both the NT drivers and the SSH code files were pre-processed manually in order to remove heap accesses, and automatically with CIL [NMRW02] in Version 1.3.6 in order to simplify the C code to facilitate the parsing. Into some of these examples artificial bugs were introduced causing specified assertions to fail. The name of these programs contains `BUG` in order to show this. All test cases are included in the CPAchecker repository together with the used configurations.

The `test_locks*` programs have less than 200 lines of code, whereas the other examples have between 700 and 3000 lines of code. These numbers include comments and empty lines and were measured after the pre-processing.

In the following, all times are shown as seconds and were rounded to three significant digits. In cases where CPAchecker did not succeed, either “>1800” or “MO” are printed for a timeout or an out of memory error respectively.

All configurations gave the correct result on all test cases, if they succeeded to terminate in the given time. That is, for all programs with `BUG` in the name a counterexample was found, and all the other programs were proved safe. Therefore, the result of the analysis is not shown in the following.

5.1 Block Size varying between SBE and LBE

The first set of evaluated configurations was created by using a new abstraction operator $\mathsf{abs^{LBE}}_k : E \times G \to \mathbb{B}$ which is defined as the disjunction of $\mathsf{abs^{LBE}}$ and abs_k. This operator returns true if either the length of the longest path represented by the path formula of the abstract state has reached a certain threshold $k \in \mathbb{N} \setminus \{0\}$ or the successor location is an abstraction location as defined by LBE.

Using another measure for the size of a path formula than the length of the longest path was also evaluated. Possible functions would include the number of Boolean operators in the formula, or the number of operands. With these the size of the new element after a merge took place would be the sum of the sizes of the old elements. However, if the program contains several successive control-flow splits and merges (as it is often the case), this number would increase exponentially. In programs like the `test_locks_*` examples, the number of operands in a formula would easily reach several hundred million. Since formulas like this contain a lot of identical sub-parts, this is neither a good approximation of the memory needed to store the formula nor the computing time needed for operations like satisfiability checks. Therefore, the length of the longest path represented by the path formula was always used as the size of a path formula. This is also more intuitive, as it is roughly identical to the length of the code in the source file.

In order to allow comparing the current results with previous ones, the test cases were run with the previous implementation of SBE and LBE (the one used by Beyer et al. [BCG+09]) and with the new implementation. The results can be seen in Table 5.1. The columns marked with "adj." are the ones produced by the implementation of Adjustable Large-Block Encoding using $\mathsf{abs^{SBE}}$ and $\mathsf{abs^{LBE}}$ respectively. Cartesian abstraction was used for SBE and Boolean abstraction was used for LBE. For "LBE (adj.)" topological sort was used as the traversal method, the other configurations used depth-first search (LBE on a pre-processed CFA does not benefit from topsort). The results show that in most cases the old implementation is faster, which is reasonable as the overhead for constructing the blocks on-the-fly is missing. However, for LBE this difference is not as large as for SBE, and there are also examples where the new

Program	SBE	SBE (adj.)	LBE	LBE (adj.)	$\max_{e \in R}(\mathsf{size}(e))$
`test_locks_5.c`	3.26	6.21	.170	.483	43
`test_locks_6.c`	3.98	13.2	.370	.398	51
`test_locks_7.c`	7.37	46.9	.237	.875	59
`test_locks_8.c`	16.3	124	.305	.437	67
`test_locks_9.c`	37.1	692	.202	.510	75
`test_locks_10.c`	109	> 1800	.266	.746	83
`test_locks_11.c`	MO	> 1800	.256	.416	91
`test_locks_12.c`	MO	MO	.248	.486	99
`test_locks_13.c`	MO	MO	.240	.769	107
`test_locks_14.c`	MO	MO	.227	.787	115
`test_locks_15.c`	> 1800	MO	.466	.896	123
`cdaudio_simpl1.cil.c`	164	1780	11.7	51.5	202
`cdaudio_simpl1_BUG.cil.c`	173	488	5.26	32.5	202
`diskperf_simpl1.cil.c`	MO	417	537	146	54
`floppy_simpl3.cil.c`	41.8	313	7.04	20.1	64
`floppy_simpl3_BUG.cil.c`	19.1	128	2.97	11.1	64
`floppy_simpl4.cil.c`	52.7	740	8.35	32.2	133
`floppy_simpl4_BUG.cil.c`	23.7	488	4.58	20.1	133
`kbfiltr_simpl1.cil.c`	7.54	38.6	1.27	2.57	53
`kbfiltr_simpl2.cil.c`	15.6	150	1.73	3.75	72
`kbfiltr_simpl2_BUG.cil.c`	8.84	59.7	1.96	2.28	72
`s3_clnt_1.cil.c`	357	> 1800	15.9	14.6	102
`s3_clnt_1_BUG.cil.c`	484	522	1.22	2.81	102
`s3_clnt_2.cil.c`	726	> 1800	12.8	35.4	102
`s3_clnt_2_BUG.cil.c`	531	469	2.12	2.06	102
`s3_clnt_3.cil.c`	594	> 1800	19.5	17.8	102
`s3_clnt_3_BUG.cil.c`	492	547	1.26	3.14	102
`s3_clnt_4.cil.c`	601	> 1800	36.6	9.59	102
`s3_clnt_4_BUG.cil.c`	462	481	2.03	2.54	102
`s3_srvr_1.cil.c`	194	> 1800	16.6	31.2	103
`s3_srvr_1_BUG.cil.c`	660	116	1.43	1.62	103
`s3_srvr_2.cil.c`	MO	> 1800	107	86.7	102
`s3_srvr_2_BUG.cil.c`	1090	23.3	1.55	2.71	102
`s3_srvr_3.cil.c`	1010	> 1800	109	14.1	102
`s3_srvr_4.cil.c`	277	> 1800	441	160	102
`s3_srvr_6.cil.c`	> 1800	> 1800	456	45.7	107
`s3_srvr_7.cil.c`	MO	> 1800	321	136	105
`s3_srvr_8.cil.c`	> 1800	> 1800	> 1800	21.2	105

Table 5.1: Comparison of SBE and LBE
$\mathsf{size}(e)$ is the length of the longest path represented by the path formula of an abstract state e, whereas R is the final reached set.

approach works better. Thus the disadvantage of the previous implementation (the missing flexibility) in general does not outweigh its sometimes better performance.

Using the $\mathsf{abs}^{\mathsf{LBE}}{}_k$ operator, one can adjust the block size freely between SBE and LBE by setting the parameter k. Firstly, it was necessary to determine meaningful values for k. For this, the maximum block size of all elements of the final reached set R was measured while using $\mathsf{abs}^{\mathsf{LBE}}$. The results are shown in the last column of Table 5.1. In the examples this value was in the range from 50 to 200. This means that the longest code part that does not contain function calls or loops has approximately that many statements. This may seem much for normal programs, but results from the CIL pre-processing before the analysis. This step simplifies complex statements by splitting them into smaller statements and introducing temporary variables, which increases the number of statements.

5.1.1 Block Sizes between 1 and 100

As only a few examples had block sizes beyond 100, the first step was to analyze sizes in the range of 1 to 100. Due to the large amount of time needed to benchmark a single configuration, only $k \in \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ was tested. For all of these configurations Boolean abstraction and topological sort were used.

The results, which are shown in Table 5.2, vary for the three groups of test cases. The `test_locks_*` examples behave as expected, producing an exponential decrease in time when the block size rises, until all examples need less than one second. This can nicely be seen when drawing the results in a diagram with an logarithmic time axis, as in Figure 5.1. While for LBE all examples take roughly the same amount of time, for smaller block sizes the larger examples take exponentially more time than the ones with less locks. For block sizes below 30 some of the larger examples do not even terminate.

For the NT drivers the results are similar, but the point from which no further performance increase is made is reached earlier. This threshold lies between 20 and 50 depending on the example. For larger block sizes, the time needed stays roughly constant, showing neither a downwards nor an upwards trend. The results for the test cases with a bug behave similar to the ones without a bug.

The SSH test cases do not show such clear results. While the times for the programs with a bug follow the same trend as the other examples, the times for the programs without an artificial bug vary widely depending on the block size. For block sizes below 50, none of the latter terminates. Even for larger blocks, some examples do

Program	$k = 1$	$k = 10$	$k = 20$	$k = 30$	$k = 40$	$k = 50$	$k = 60$	$k = 70$	$k = 80$	$k = 90$	$k = 100$	LBE
`test_locks_5.c`	6.06	3.42	1.02	1.29	.367	.695	.397	.292	.587	.468	.507	.483
`test_locks_6.c`	20.2	3.03	1.90	1.36	.690	.334	.527	.428	.637	.790	.323	.398
`test_locks_7.c`	48.4	5.71	1.30	3.26	.516	1.06	.800	.326	.591	.355	.807	.875
`test_locks_8.c`	220	25.8	3.82	1.86	1.20	1.27	.414	.392	.670	.575	.680	.437
`test_locks_9.c`	341	67.7	12.4	6.97	1.67	1.63	.543	.454	.551	.667	.705	.510
`test_locks_10.c`	> 1800	109	7.53	6.59	3.79	1.24	.679	.588	.805	.845	.993	.746
`test_locks_11.c`	MO	244	26.7	4.71	6.73	1.71	.906	.992	1.20	.905	.418	.416
`test_locks_12.c`	MO	> 1800	88.5	20.6	3.08	5.31	1.32	1.04	.995	1.35	.728	.486
`test_locks_13.c`	> 1800	MO	134	71.5	7.12	2.78	2.29	1.48	1.77	1.05	1.09	.769
`test_locks_14.c`	MO	MO	> 1800	580	19.6	17.6	4.61	2.25	2.07	1.13	.915	.787
`test_locks_15.c`	MO	> 1800	> 1800	> 1800	32.2	22.3	23.1	5.56	2.71	2.46	1.40	.896
`cdaudio_simpl1.cil.c`	MO	210	119	51.9	52.9	54.5	49.0	52.6	58.1	53.8	53.5	51.5
`cdaudio_simpl1_BUG.cil.c`	478	106	151	32.6	36.7	38.6	32.7	35.5	40.0	33.4	31.9	32.5
`diskperf_simpl1.cil.c`	> 1800	855	155	171	163	168	158	152	154	146	167	146
`floppy_simpl3.cil.c`	498	80.5	23.6	19.3	25.5	23.1	20.0	21.0	21.1	19.8	17.9	20.1
`floppy_simpl3_BUG.cil.c`	168	45.9	13.9	12.1	13.9	11.3	11.2	9.38	9.11	10.5	10.4	11.1
`floppy_simpl4.cil.c`	962	212	54.0	28.8	41.4	39.2	35.2	31.7	32.6	32.8	44.6	32.2
`floppy_simpl4_BUG.cil.c`	352	150	39.0	16.9	30.4	31.3	26.1	21.3	22.2	23.3	23.1	20.1
`kbfiltr_simpl1.cil.c`	30.6	10.1	3.72	2.66	3.28	2.82	2.15	2.49	1.83	1.89	2.81	2.57
`kbfiltr_simpl2.cil.c`	114	59.1	10.2	5.26	5.90	7.93	4.12	4.56	4.19	4.67	3.94	3.75
`kbfiltr_simpl2_BUG.cil.c`	42.2	16.5	3.25	4.16	3.22	3.37	2.89	3.22	2.19	2.36	2.27	2.28
`s3_clnt_1.cil.c`	MO	MO	MO	MO	MO	41.3	27.8	13.4	10.8	445	45.0	14.6
`s3_clnt_1_BUG.cil.c`	MO	67.5	20.4	8.78	11.8	3.82	2.39	2.25	2.16	7.87	4.19	2.81
`s3_clnt_2.cil.c`	MO	MO	MO	MO	MO	34.4	45.0	16.2	12.8	569	49.1	35.4
`s3_clnt_2_BUG.cil.c`	MO	135	26.6	14.2	10.2	3.93	3.05	1.94	2.59	6.47	3.58	2.06
`s3_clnt_3.cil.c`	MO	MO	MO	MO	MO	45.7	238	309	24.6	MO	36.4	17.8
`s3_clnt_3_BUG.cil.c`	MO	55.3	18.6	19.6	6.20	3.62	2.72	3.27	1.93	9.87	3.45	3.14
`s3_clnt_4.cil.c`	MO	MO	MO	MO	MO	38.1	17.7	24.2	9.53	441	28.4	9.59
`s3_clnt_4_BUG.cil.c`	994	78.0	33.8	15.2	5.42	3.73	2.35	1.86	2.68	8.91	3.64	2.54
`s3_srvr_1.cil.c`	MO	MO	MO	MO	MO	43.7	MO	712	113	MO	47.8	31.2
`s3_srvr_1_BUG.cil.c`	163	14.9	9.44	2.03	3.01	1.49	2.64	2.32	2.35	5.22	1.47	1.62
`s3_srvr_2.cil.c`	MO	MO	MO	MO	MO	462	33.2	MO	340	MO	98.5	86.7
`s3_srvr_2_BUG.cil.c`	21.8	60.4	6.18	2.72	4.80	2.65	1.09	2.03	1.61	5.00	3.32	2.71
`s3_srvr_3.cil.c`	MO	MO	MO	MO	MO	32.9	11.5	31.1	24.7	MO	MO	14.1
`s3_srvr_4.cil.c`	MO	MO	MO	MO	MO	325	56.4	12.1	22.0	MO	45.6	160
`s3_srvr_6.cil.c`	MO	MO	MO	MO	MO	MO	83.8	638	MO	50.8	MO	45.7
`s3_srvr_7.cil.c`	MO	MO	MO	MO	MO	MO	133	458	MO	MO	315	136
`s3_srvr_8.cil.c`	MO	MO	MO	MO	MO	MO	18.9	42.7	26.8	155	565	21.2

Table 5.2: Results for block sizes 1 to 100, Boolean abstraction, topological sort

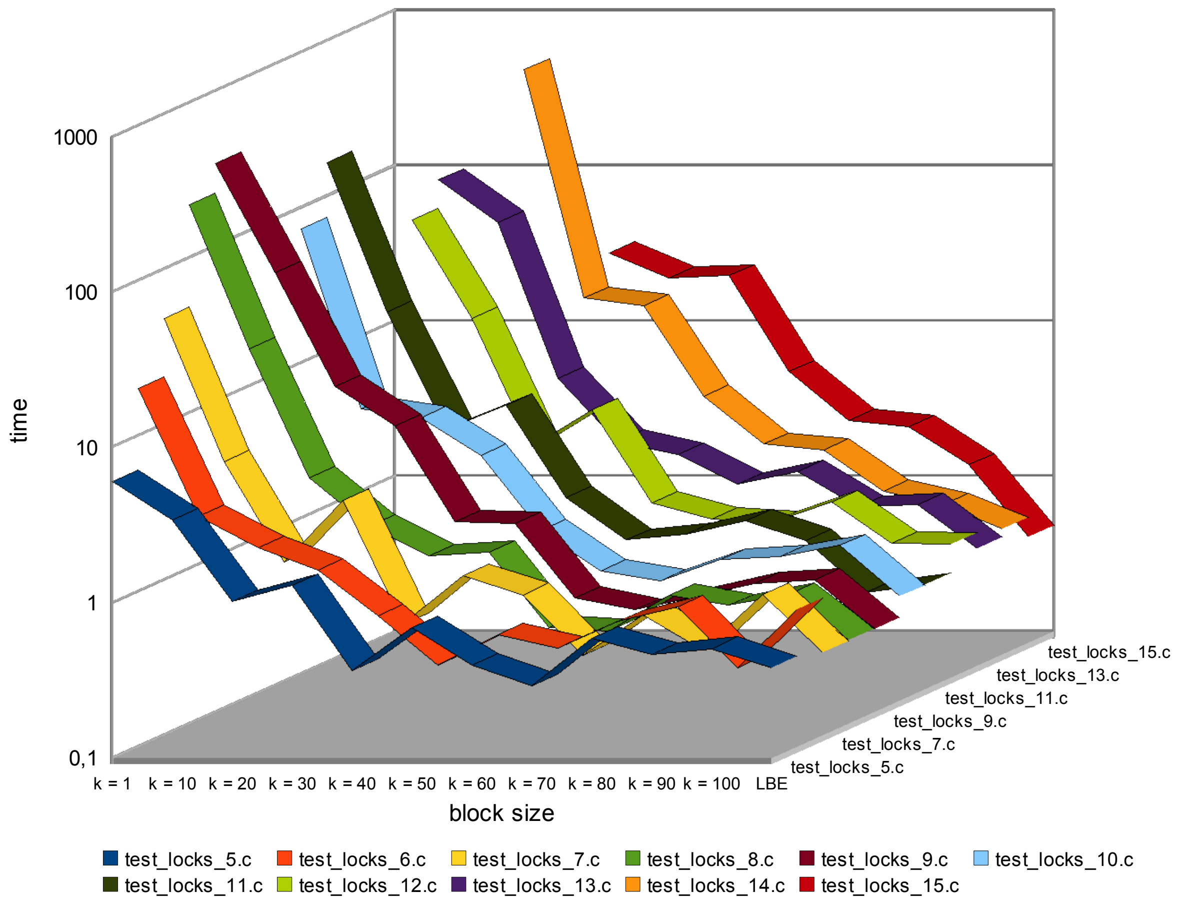


Figure 5.1: Results for block sizes 1 to 100, Boolean abstraction, topological sort

not terminate although configurations with smaller blocks succeed in analyzing them. Even with a block size of 100 two examples fail to be analyzed, although their maximum block size when analyzed with LBE only reaches 102 and 107 respectively. The only configuration which works for all examples is LBE.

Taken together, out of these configurations with Boolean abstraction and topological sort, LBE is clearly the best configuration as expected. Not only is it the only one succeeding on all examples used, it is also the fastest for almost all examples.

5.1.2 Block Sizes between 1 and 10

For very small block sizes, Cartesian abstraction was expected to be able to verify the programs. Therefore, several benchmarks were run with $k \in \{1, \ldots, 10\}$. Three configurations were used, namely Cartesian abstraction with DFS (Table 5.3), Cartesian abstraction with topological sort (Table 5.4) and Boolean abstraction with topological sort (Table 5.5). The last one was included because it is the same configuration that was used in the previous section.

The first notable result here is that Cartesian abstraction with topological sort manages to verify more examples if the block size is lower, succeeding at the smallest 4 `test_locks_*` examples and all the NT driver examples with $k = 1$, but at no `test_locks_*` and only 2 driver programs with $k = 10$. The reason is that the complexity of the path formulas rises when abstract states are merged and one formula represents several paths of the program. Then the path formulas are not longer conjunctions of predicates, but arbitrary Boolean formulas containing predicates that are combined with “and” and “or” without any restriction. Cartesian abstraction often is too imprecise for this because it considers only one predicate at a time. As topological sort is used to maximize the number of merges that occur during the analysis, this explains why Cartesian abstraction performs so badly with this state-space traversal order. Due to a limitation of CPAchecker it is not detected when the analysis is too imprecise for the analyzed program. Instead there is an endless loop in which the same spurious counterexample is found over and over again. Therefore, these cases show up as a timeout or an out-of-memory error in the result tables.

Table 5.3 shows that Cartesian abstraction performs better when used together with DFS. It manages to verify almost all of the `test_locks_*` examples if the block size threshold is at least 8. Only for the driver programs the results are similar to that of Cartesian abstraction with topological sort, also getting worse at a higher k. This is explicable by the greater complexity of the control flow in these programs compared with the other examples.

The results for Boolean abstraction shown in Table 5.5 are those that could be expected from the results in Table 5.2, confirming the fact that larger block sizes are better in terms of performance. While it was expected that Boolean abstraction is slower than Cartesian abstraction as long as the latter is precise enough, this is not always true. The advantage of being able to work with more complex formulas (and thus with topological sort and merging, which reduces the number of abstractions) seems to at least sometimes outweigh the costlier abstraction computations. Boolean abstraction with DFS was not tested due to time reasons. It is expected that this configuration would be the slowest of all four as the results for Cartesian abstraction show that DFS mostly does not need the better precision provided by Boolean abstraction.

The overall result of this section is clearly that there is no other configuration with a performance similar to that of LBE. In the tested range, larger blocks are always faster than smaller blocks. While this effect flattens above $k = 50$, some examples do still benefit from a further increase in the size of the blocks, so it seems to be worth evaluating block sizes beyond those that LBE produces. Cartesian abstraction cannot be seen as a usable configuration. For $k \leq 6$ a large number of test cases fail because the analysis would take too much time or memory, and for larger k the imprecision that is inherent to this method impedes its application.

5.2 Block Sizes larger than LBE

As the previous section shows that smaller block sizes are slower than LBE, it is interesting to see how the analysis performs for larger block sizes. For this, abs_k was used as the abstraction operator together with Boolean abstraction and topological sort. The threshold k was taken from $\{50, 100, 150, 200, 250, 300\}$ in order to test configurations that produce blocks larger than those of LBE for most of the examples.

For the NT drivers, the results are shown in Table 5.6. For all other examples the analysis failed because of too few memory or a timeout. The reason for this is that these programs consist of a single large loop. With abs_k as the abstraction operator the analysis fails to detect in a timely manner that the abstract states inside of the loop are covered after the second iteration. For a new abstract state, the check if it is covered is only made if it was produced by an abstraction. This is because only for a pair of such nodes it is possible to do this by comparing only their abstraction formulas, which can be done very fast. Checking if a symbolic formula (such as a path formula) entails another formula is too expensive to do this for all new abstract states. Additionally, as the program counter is modeled explicitly by the analysis, an abstract state can only be covered by another abstract state that belongs to the same location.

Furthermore, a loop in the analyzed program is unrolled in the ART until at least one of two conditions is true: Either the path has become infeasible (for example if a counter variable exceeds the upper limit) or no new information has been added to the abstract states in the last iteration. For the first condition, it is necessary that there is a predicate in the precision that is related to the loop condition. This is often not the case. Especially with lazy abstraction it is never the case in the first iteration of the abstraction-refinement-loop as the initial precision is empty. For the second condition, it is necessary that at least one abstract state in the loop is detected to be covered by another abstract state that has been generated before.

Now if the threshold k is larger than the path through a loop of the program, but it is not a multiple of the length s of the longest path through the loop, there will be no

Program	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
`test_locks_5.c`	6.36	5.01	5.97	5.52	5.12	5.29	4.06	6.14	3.36	3.42
`test_locks_6.c`	13.1	10.2	11.2	10.8	12.1	11.1	11.7	6.16	11.1	4.75
`test_locks_7.c`	34.8	22.8	28.5	25.3	32.7	31.5	31.6	17.3	23.1	31.0
`test_locks_8.c`	102	42.8	51.6	66.1	72.9	40.2	50.1	37.3	39.0	29.8
`test_locks_9.c`	298	95.9	118	190	145	92.8	95.6	126	106	73.7
`test_locks_10.c`	1250	271	336	440	474	185	233	147	176	274
`test_locks_11.c`	> 1800	883	823	1270	1180	702	441	467	265	735
`test_locks_12.c`	> 1800	> 1800	> 1800	> 1800	> 1800	1280	1450	599	529	686
`test_locks_13.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	1220	1560	1060
`test_locks_14.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`test_locks_15.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`cdaudio_simpl1.cil.c`	MO	MO	1020	MO	898	MO	MO	336	493	617
`cdaudio_simpl1_BUG.cil.c`	158	118	94.0	71.0	MO	MO	MO	MO	MO	MO
`diskperf_simpl1.cil.c`	MO	MO	897	MO	302	293	213	MO	101	62.0
`floppy_simpl3.cil.c`	559	MO	882	742	610	MO	MO	MO	MO	MO
`floppy_simpl3_BUG.cil.c`	75.8	78.5	74.9	MO	MO	MO	MO	MO	MO	MO
`floppy_simpl4.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`floppy_simpl4_BUG.cil.c`	77.4	81.0	78.7	47.8	MO	MO	MO	MO	MO	MO
`kbfiltr_simpl1.cil.c`	48.2	39.9	30.9	.608	.568	1.33	1.24	1.13	1.01	.577
`kbfiltr_simpl2.cil.c`	128	124	115	42.2	39.5	30.4	12.3	32.9	21.5	11.2
`kbfiltr_simpl2_BUG.cil.c`	156	143	90.8	130	105	105	53.2	99.5	50.0	69.9
`s3_clnt_1.cil.c`	MO	> 1800	> 1800	> 1800	> 1800	821	492	265	441	636
`s3_clnt_1_BUG.cil.c`	667	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`s3_clnt_2.cil.c`	MO	MO	> 1800	538	> 1800	322	> 1800	204	393	523
`s3_clnt_2_BUG.cil.c`	677	299	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`s3_clnt_3.cil.c`	MO	> 1800	> 1800	MO	> 1800	552	> 1800	184	653	525
`s3_clnt_3_BUG.cil.c`	653	1370	> 1800	MO	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`s3_clnt_4.cil.c`	MO	> 1800	MO	726	> 1800	330	776	246	445	495
`s3_clnt_4_BUG.cil.c`	646	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`s3_srvr_1.cil.c`	> 1800	> 1800	> 1800	214	1510	> 1800	864	29.7	304	3.93
`s3_srvr_1_BUG.cil.c`	42.2	45.8	16.8	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`s3_srvr_2.cil.c`	MO	> 1800	> 1800	> 1800	189	> 1800	MO	378	178	327
`s3_srvr_2_BUG.cil.c`	35.6	165	19.2	> 1800	> 1800	> 1800	MO	> 1800	> 1800	> 1800
`s3_srvr_3.cil.c`	> 1800	> 1800	> 1800	18.1	714	> 1800	56.4	685	> 1800	1080
`s3_srvr_4.cil.c`	> 1800	> 1800	> 1800	87.7	1130	> 1800	2.33	414	231	351
`s3_srvr_6.cil.c`	> 1800	> 1800	> 1800	732	1660	> 1800	4.53	> 1800	180	> 1800
`s3_srvr_7.cil.c`	> 1800	> 1800	> 1800	> 1800	998	> 1800	10.3	519	258	> 1800
`s3_srvr_8.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	3.90	> 1800	4.91	> 1800

Table 5.3: Results for block sizes 1 to 10, Cartesian abstraction, DFS

Program	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
`test_locks_5.c`	6.21	20.0	23.8	> 1800	MO	MO	MO	MO	MO	MO
`test_locks_6.c`	13.2	117	> 1800	> 1800	> 1800	MO	MO	MO	MO	MO
`test_locks_7.c`	46.9	> 1800	> 1800	> 1800	> 1800	MO	MO	MO	MO	MO
`test_locks_8.c`	124	> 1800	> 1800	> 1800	> 1800	MO	MO	MO	MO	MO
`test_locks_9.c`	692	> 1800	> 1800	> 1800	> 1800	MO	MO	MO	MO	MO
`test_locks_10.c`	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO	MO	MO
`test_locks_11.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	MO	MO	MO
`test_locks_12.c`	MO	> 1800	> 1800	> 1800	> 1800	MO	MO	MO	> 1800	MO
`test_locks_13.c`	MO	> 1800	> 1800	> 1800	> 1800	> 1800	MO	MO	MO	MO
`test_locks_14.c`	MO	MO	> 1800	> 1800	> 1800	> 1800	MO	MO	MO	MO
`test_locks_15.c`	MO	MO	> 1800	> 1800	> 1800	> 1800	> 1800	MO	MO	MO
`cdaudio_simpl1.cil.c`	1780	1190	1100	766	453	MO	MO	340	MO	MO
`cdaudio_simpl1_BUG.cil.c`	488	456	629	476	256	265	225	265	148	124
`diskperf_simpl1.cil.c`	417	293	222	144	161	MO	65.4	MO	MO	MO
`floppy_simpl3.cil.c`	313	315	349	323	152	260	MO	MO	MO	MO
`floppy_simpl3_BUG.cil.c`	128	271	209	182	74.0	122	43.2	MO	MO	MO
`floppy_simpl4.cil.c`	740	1100	1060	871	541	356	MO	MO	MO	MO
`floppy_simpl4_BUG.cil.c`	488	676	712	584	341	250	141	MO	MO	MO
`kbfiltr_simpl1.cil.c`	38.6	29.5	25.5	18.7	19.4	MO	6.87	MO	MO	MO
`kbfiltr_simpl2.cil.c`	150	122	141	94.1	82.4	MO	30.4	MO	MO	MO
`kbfiltr_simpl2_BUG.cil.c`	59.7	26.7	26.3	17.8	18.9	13.9	6.33	14.9	10.3	17.6
`s3_clnt_1.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO
`s3_clnt_1_BUG.cil.c`	522	529	189	240	64.6	63.1	MO	59.7	87.0	142
`s3_clnt_2.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	> 1800
`s3_clnt_2_BUG.cil.c`	469	241	420	303	160	144	MO	MO	81.2	89.3
`s3_clnt_3.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	> 1800
`s3_clnt_3_BUG.cil.c`	547	223	279	462	94.2	221	156	MO	125	55.8
`s3_clnt_4.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	> 1800
`s3_clnt_4_BUG.cil.c`	481	253	258	489	147	42.1	> 1800	MO	66.7	93.0
`s3_srvr_1.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO	MO
`s3_srvr_1_BUG.cil.c`	116	45.8	25.7	393	> 1800	25.6	MO	57.9	MO	MO
`s3_srvr_2.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO	> 1800
`s3_srvr_2_BUG.cil.c`	23.3	298	66.4	112	> 1800	24.9	> 1800	83.4	MO	51.6
`s3_srvr_3.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO	MO
`s3_srvr_4.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO	MO
`s3_srvr_6.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO	MO
`s3_srvr_7.cil.c`	> 1800	> 1800	MO	> 1800	> 1800	> 1800	MO	> 1800	MO	MO
`s3_srvr_8.cil.c`	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	MO	MO

Table 5.4: Results for block sizes 1 to 10, Cartesian abstraction, topological sort

Program	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
`test_locks_5.c`	6.06	18.0	24.5	11.2	6.05	3.26	3.09	3.27	2.48	3.42
`test_locks_6.c`	20.2	MO	> 1800	119	15.5	8.83	11.2	3.80	2.47	3.03
`test_locks_7.c`	48.4	MO	MO	648	83.5	91.8	12.6	13.0	21.3	5.71
`test_locks_8.c`	220	MO	> 1800	MO	180	291	47.0	50.3	24.2	25.8
`test_locks_9.c`	341	> 1800	> 1800	> 1800	1350	417	304	49.8	38.8	67.7
`test_locks_10.c`	> 1800	> 1800	> 1800	> 1800	> 1800	1220	587	1480	271	109
`test_locks_11.c`	MO	> 1800	> 1800	> 1800	> 1800	> 1800	1550	1270	1460	244
`test_locks_12.c`	MO	MO	> 1800	> 1800	> 1800	> 1800	> 1800	MO	MO	> 1800
`test_locks_13.c`	> 1800	> 1800	> 1800	> 1800	> 1800	MO	> 1800	> 1800	MO	MO
`test_locks_14.c`	MO	MO	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	MO
`test_locks_15.c`	MO	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800	> 1800
`cdaudio_simpl1.cil.c`	MO	> 1800	> 1800	587	580	624	366	516	246	210
`cdaudio_simpl1_BUG.cil.c`	478	358	526	391	308	226	192	250	133	106
`diskperf_simpl1.cil.c`	> 1800	> 1800	> 1800	423	> 1800	> 1800	237	59.2	169	855
`floppy_simpl3.cil.c`	498	436	404	237	115	440	73.4	68.1	65.1	80.5
`floppy_simpl3_BUG.cil.c`	168	216	213	189	82.1	81.1	30.8	31.2	34.7	45.9
`floppy_simpl4.cil.c`	962	897	MO	771	514	618	141	294	167	212
`floppy_simpl4_BUG.cil.c`	352	585	633	448	320	202	121	199	128	150
`kbfiltr_simpl1.cil.c`	30.6	18.7	20.0	17.6	15.3	17.8	5.85	3.94	7.25	10.1
`kbfiltr_simpl2.cil.c`	114	96.8	122	77.7	72.5	56.0	24.1	18.9	39.3	59.1
`kbfiltr_simpl2_BUG.cil.c`	42.2	16.2	17.5	16.8	15.6	11.2	5.24	12.8	8.64	16.5
`s3_clnt_1.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_clnt_1_BUG.cil.c`	MO	814	MO	436	48.9	119	201	97.9	175	67.5
`s3_clnt_2.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_clnt_2_BUG.cil.c`	MO	536	375	511	257	179	57.0	61.3	115	135
`s3_clnt_3.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_clnt_3_BUG.cil.c`	MO	593	592	809	229	172	83.7	82.8	169	55.3
`s3_clnt_4.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_clnt_4_BUG.cil.c`	994	324	206	845	83.2	47.4	138	31.0	78.2	78.0
`s3_srvr_1.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_srvr_1_BUG.cil.c`	163	52.6	138	136	24.8	42.5	46.4	36.8	20.8	14.9
`s3_srvr_2.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_srvr_2_BUG.cil.c`	21.8	378	83.1	224	17.8	24.4	154	64.9	16.6	60.4
`s3_srvr_3.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_srvr_4.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_srvr_6.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_srvr_7.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO
`s3_srvr_8.cil.c`	MO	MO	MO	MO	MO	MO	MO	MO	MO	MO

Table 5.5: Results for block sizes 1 to 10, Boolean abstraction, topological sort

Program	$k = 50$	$k = 100$	$k = 150$	$k = 200$	$k = 250$	$k = 300$	LBE
`cdaudio_simpl1.cil.c`	223	47.2	18.9	14.8	> 1800	> 1800	51.5
`cdaudio_simpl1_BUG.cil.c`	158	97.6	26.6	18.7	> 1800	> 1800	32.5
`diskperf_simpl1.cil.c`	3.94	3.44	2.68	1.65	2.42	2.51	146
`floppy_simpl3.cil.c`	10.8	2.10	2.75	1.11	1.43	2.50	20.1
`floppy_simpl3_BUG.cil.c`	13.9	1.86	3.17	1.22	.891	1.90	11.1
`floppy_simpl4.cil.c`	3.06	2.08	4.07	.952	1.41	1.99	32.2
`floppy_simpl4_BUG.cil.c`	2.56	1.69	4.05	1.61	1.81	2.74	20.1
`kbfiltr_simpl1.cil.c`	2.97	1.79	.761	1.07	1.38	1.31	2.57
`kbfiltr_simpl2.cil.c`	5.39	2.14	2.16	1.69	1.39	1.65	3.75
`kbfiltr_simpl2_BUG.cil.c`	3.28	2.03	2.11	1.44	1.39	2.32	2.28

Table 5.6: Results for block sizes 50 to 300 with abs_k

coverage check as long as no two nodes for the same location have been created. In the worst case, this can take $k \cdot s$ steps and s abstraction computations. For $k = s$, only one iteration and two abstraction computations would have been needed. Therefore it is not advisable to use abs_k with larger values of k if the program contains large loops and no predicates are tracked for the loop condition.

However, as the results for the driver examples show, it may be worth to use abs_k for programs without large loops, as the analysis is sometimes a lot faster than LBE. The best block size depends strongly on the analyzed program, so no general suggestion can be given. The highly differing results for $\mathsf{abs}^{\mathsf{LBE}}{}_k$ (see Table 5.2) and abs_k (see Table 5.6) with the same k show that with LBE a lot of blocks are actually smaller than 50 although the maximum block size is always larger than that for these programs.

5.2.1 Loop Unrolling

In order to be able to use larger blocks for the examples with large loops in spite of the encountered problem, abs_k was combined with $\mathsf{abs}^{\mathsf{LBE}}$ so that abstractions would be computed only at the abstraction locations that LBE uses (especially loop heads), but with some loop iterations being unrolled. This can be achieved by using the conjunction of abs_k and $\mathsf{abs}^{\mathsf{LBE}}$ as the abstraction operator.

In Table 5.7 the results are shown for the SSH test cases with the threshold k taken from $\{50, 100, 150, 200, 250, 300\}$. For the other examples there is no difference in performance to the LBE results. The reason for this is that the loop in the `test_locks*` programs is not relevant for the safety properties that the analysis tries to verify. Therefore, only one iteration of the loop is analyzed in the LBE configuration and no performance benefit can be gained by unrolling. There is also no measurable slowdown because there are only very few abstraction computations necessary, and these do not need much more time if the path formula is for example twice as large as with LBE (if

Program	$k = 50$	$k = 100$	$k = 150$	$k = 200$	$k = 250$	$k = 300$	LBE
`s3_clnt_1.cil.c`	17.4	9.54	432	527	829	> 1800	14.6
`s3_clnt_1_BUG.cil.c`	1.91	1.99	4.64	5.68	4.38	8.33	2.81
`s3_clnt_2.cil.c`	35.2	34.2	MO	18.7	1090	> 1800	35.4
`s3_clnt_2_BUG.cil.c`	2.88	2.03	4.43	4.71	7.94	9.17	2.06
`s3_clnt_3.cil.c`	19.2	28.6	35.7	> 1800	> 1800	916	17.8
`s3_clnt_3_BUG.cil.c`	3.69	2.62	5.18	5.13	6.82	9.60	3.14
`s3_clnt_4.cil.c`	8.58	29.5	1200	> 1800	1290	> 1800	9.59
`s3_clnt_4_BUG.cil.c`	3.08	2.83	4.54	4.51	5.85	5.67	2.54
`s3_srvr_1.cil.c`	36.1	24.1	1560	MO	74.4	> 1800	31.2
`s3_srvr_1_BUG.cil.c`	1.73	2.33	3.27	3.95	3.24	3.66	1.62
`s3_srvr_2.cil.c`	91.2	18.5	24.1	MO	> 1800	MO	86.7
`s3_srvr_2_BUG.cil.c`	2.17	1.96	2.34	2.80	2.96	2.91	2.71
`s3_srvr_3.cil.c`	15.5	20.5	MO	303	> 1800	> 1800	14.1
`s3_srvr_4.cil.c`	181	453	MO	> 1800	MO	613	160
`s3_srvr_6.cil.c`	51.4	485	112	MO	MO	MO	45.7
`s3_srvr_7.cil.c`	146	23.1	101	607	> 1800	> 1800	136
`s3_srvr_8.cil.c`	21.6	15.1	86.6	1330	1000	276	21.2

Table 5.7: Results for LBE with unrolling of loops

two iterations were unrolled). The NT drivers' files contain at most one small loop, so there is no difference in performance for them, either.

For the SSH test cases, the results are mixed. Some configurations provide better performance than LBE for a few programs. With almost all configurations, however, there are some examples where much more time is needed, or the analysis even fails to terminate. An extreme case is `s3_clnt_4.cil.c`. LBE needs less than 10 seconds, but with $k >= 150$ more than 1000 seconds are needed. Similar examples were observed during implementation. Even a simple artificial example (shorter than the presented `lock-loop.c`) that LBE manages to verify in under a second could take over 500 seconds when two iterations of the loop were unrolled.

The main cause for this are the interpolants generated by the SMT-solver. In these cases, the path formula is too complex to generate interpolants efficiently. The interpolation takes very long, and the resulting formulas are very large. Even for the smallest tested example, a single interpolant could be several Megabytes large when represented as a DAG. If such a formula was unrolled into a normal string representation consisting of predicates concatenated with "&" and "|", it could reach up to 170 MB even though the originating program contained only a small loop with 20 iterations and the threshold k was set to 10.

To see if this can be done better by other tools, a different SMT-solver was tried, namely CSIsat [BZM08]. While it managed to produce much better interpolants for the small artificial example, it was too slow to be used with the examples used for the benchmarks. One reason for this is that MathSAT is used as a library in CPAchecker, so calls to the solver practically generate no overhead. CSIsat in contrast is a stand-alone tool. Therefore, a new process had to be started whenever interpolants had to be generated. Also, CPAchecker does not store the formulas it uses on its own but delegates this task to MathSAT. Thus, the formulas had to be converted between the formats of MathSAT and CSIsat before and after the interpolant generation. It was not possible to better integrate CPAchecker and CSIsat due to the large dependency of predicate abstraction in CPAchecker on MathSAT, but this might be feasible in the future.

In general, loop unrolling is currently not advisable. However, work should be done to generate better interpolants, either by enhancing MathSAT or by integrating CSIsat more tightly so that its usage in CPAchecker becomes faster. As the results look promising aside from the cases with extreme problems, loop unrolling might be a future way to reduce the amount of abstraction computations.

5.2.2 Function Inlining

Yet another way to increase the block size was evaluated. The previous implementation of LBE forces an abstraction on every function call and return, as explained in Chapter 4. In order to see if this is really necessary, this restriction was changed to be optional. The variant without these abstractions is called LBE with function inlining, as it behaves as if a pre-processor would have inlined all function calls in the program. This is not possible for recursive functions, and similarly LBE with function inlining will not work if recursive functions exist in the program. In this case an abstraction operator with an additional threshold that forces an abstraction after a certain path length has to be used. However, as CPAchecker currently contains no support for recursive functions anyway, this was not necessary here.

Table 5.8 compares the results of LBE with and without inlining. For the `test_locks_*` examples there is no difference, as these do not contain function calls. Therefore, they were omitted from the table. The results are inconsistent. Some test cases benefit from inlining and are notably faster, but there is a massive performance decrease for some other examples. A small performance increase was expected due to the reduced number of abstraction computations. For the results that got slower the increased complexity of the path formulas is probably too much for the used SMT-solver, similar to the results that are obtained when unrolling of loops is enabled. Therefore, it is better to not use this variant as long as interpolant generation does not work better.

Program	LBE	LBE with inlining
cdaudio_simpl1.cil.c	51.5	51.7
cdaudio_simpl1_BUG.cil.c	32.5	16.4
diskperf_simpl1.cil.c	146	273
floppy_simpl3.cil.c	20.1	21.9
floppy_simpl3_BUG.cil.c	11.1	7.08
floppy_simpl4.cil.c	32.2	23.9
floppy_simpl4_BUG.cil.c	20.1	8.83
kbfiltr_simpl1.cil.c	2.57	1.77
kbfiltr_simpl2.cil.c	3.75	3.32
kbfiltr_simpl2_BUG.cil.c	2.28	.839
s3_clnt_1.cil.c	14.6	19.7
s3_clnt_1_BUG.cil.c	2.81	2.46
s3_clnt_2.cil.c	35.4	14.2
s3_clnt_2_BUG.cil.c	2.06	4.03
s3_clnt_3.cil.c	17.8	12.7
s3_clnt_3_BUG.cil.c	3.14	2.58
s3_clnt_4.cil.c	9.59	13.3
s3_clnt_4_BUG.cil.c	2.54	3.58
s3_srvr_1.cil.c	31.2	7.30
s3_srvr_1_BUG.cil.c	1.62	2.54
s3_srvr_2.cil.c	86.7	233
s3_srvr_2_BUG.cil.c	2.71	2.46
s3_srvr_3.cil.c	14.1	36.9
s3_srvr_4.cil.c	160	14.6
s3_srvr_6.cil.c	45.7	>1800
s3_srvr_7.cil.c	136	72.5
s3_srvr_8.cil.c	21.2	19.6

Table 5.8: Results for LBE with function inlining

6 Conclusion and Future Work

Currently no other configuration provides results as good as those of LBE with Boolean abstraction. Cartesian abstraction is either too slow or too imprecise, and small block sizes are generally much slower than larger ones. However it is not beneficial to use blocks that are larger than those produced by LBE, either. All such configurations were too slow or even failed to terminate in a significant number of examples. The reason is the greater complexity of the path formulas, which seems to be too much for the used SMT-solver.

Thus several possibilities for future work arise. Perhaps a different abstraction operator that adjusts the block size more intelligently than a simple threshold would give better results. The unrolling of loops seems to be promising, but is currently restricted due to the low quality of the interpolants found by the SMT-solver used. It may be worth to extend the experiments with other SMT-solvers to see if better interpolants can be found. These tools would have to be tightly integrated into the model checker in order to provide reasonable performance. With loop unrolling and very large blocks, predicate abstraction would become similar to Bounded Model Checking [BCCZ99], so strategies for handling large formulas from BMC could be used. The CPA could be extended to allow satisfiability checks of symbolic formulas instead of using abstraction, creating an analysis that would be able to behave not only like a precise model checker, but also like a bounded model checker. Another analysis that is based on lazy abstraction and interpolants but does not compute predicate abstractions was presented by McMillan [McM06]. This approach could be integrated, too. The convergence of all these different methods would yield great possibilites for comparison. Additionally, using a configurable analysis would make it possible to use new configurations not yet presented by combining strategies from different approaches.

Another strategy for faster analysis of loops is to replace them by their closed form, if one can be found. This has been presented in [KW10] and similar techniques could be integrated into the presented CPA, too. Compared with SBE, the use of Large-Block Encoding simplifies this, because a single path formula representing the whole body of a loop is already created. This formula could be analyzed to find loop invariants.

As explained in Chapter 4, currently the new predicates found during refinement are added only to the abstract states along the infeasible path. However, if there are several similar paths through the program, it may be worth adding the predicates to other paths as well, saving the need for separate refinement steps for each path. Heuristics could be implemented that share the predicates of similar paths. With the possibility for dynamic precision adjustment [BHT08], the CPA framework already provides a convenient method to do this kind of precision adjustment.

Most analysis that are based on predicates focus on precision rather than on efficiency. As the CPA framework makes it easy to configure the precision of an analysis by choosing an appropriate merge operator, it might be interesting to evaluate if less precise configurations (like using $\mathsf{merge}^{\mathrm{join}}$) still provide useful results and are notably faster.

All these options could probably be easily integrated into the framework for Adjustable Large-Block Encoding. This unification of concepts would create a great tool for experimenting with new approaches to software model checking based on predicates.

Bibliography

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. *Symbolic Model Checking without BDDs.* Proceedings of the 5th International Conference on Tools and Algorithms for the Analysis and Construction of Systems (TACAS 1999), LNCS 1579, pages 193–207. Springer, March 1999.

[BCF+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. *The* MATHSAT *4 SMT Solver.* Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), LNCS 5123, pages 299–303. Springer, July 2008.

[BCG+09] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. *Software Model Checking via Large-Block Encoding.* Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009), pages 25–32. IEEE Computer Society, November 2009.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. *The Software Model Checker* BLAST*: Applications to Software Engineering.* International Journal on Software Tools for Technology Transfer (STTT), volume 9, number 5-6, pages 505–525. Springer, October 2007.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis.* Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007), LNCS 4590, pages 504–518. Springer, July 2007.

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Program Analysis with Dynamic Precision Adjustment.* Proceedings of the 23rd

IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pages 29–38. IEEE Computer Society, September 2008.

[BK09] Dirk Beyer and M. Erkan Keremoglu. CPAchecker*: A Tool for Configurable Software Verification.* Technical Report SFU-CS-2009-02, Simon Fraser University (SFU), January 2009.

[BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. *Boolean and Cartesian Abstractions for Model Checking C Programs.* Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), LNCS 2031, pages 268–283. Springer, April 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. *The* Slam *project: Debugging system software via static analysis.* Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2002), pages 1–3. ACM, January 2002.

[BZM08] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat*: Interpolation for LA+EUF.* Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008), LNCS 5123, pages 304–308. Springer, July 2008.

[CCF$^+$07] Roberto Cavada, Allesandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R. K. Shyamasundar. *Computing Predicate Abstractions by Integrating BDDs and SMT Solvers.* Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2007), pages 69–76. IEEE Computer Society, November 2007.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. *Efficiently computing static single assignment form and the control dependence graph.* ACM Transactions on Programming Languages and Systems (TOPLAS), volume 13, number 4, pages 451–490. ACM, October 1991.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Counterexample-guided abstraction refinement for symbolic model* *checking.* Journal of the ACM, volume 50, number 5, pages 752–794. ACM, September 2003.

[CGS08] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. *Efficient Interpolant Generation in Satisfiability Modulo Theories.* Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), LNCS 4963, pages 397–412. Springer, April 2008.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. *A Tool for Checking ANSI-C Programs.* Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), LNCS 2988, pages 168–176. Springer, March 2004.

[CKSY05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SatAbs*: SAT-Based Predicate Abstraction for ANSI-C.* Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), LNCS 3440, pages 570–574. Springer, April 2005.

[Cra57] William Craig. *Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory.* The Journal of Symbolic Logic, volume 22, number 3, pages 269–285. Association for Symbolic Logic, September 1957.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. *A Survey of Automated Techniques for Formal Software Verification.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, volume 27, number 7, pages 1165–1178. IEEE Circuits and Systems Society, July 2008.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley, November 1994.

[GS97] Susanne Graf and Hassen Saïdi. *Construction of abstract state graphs with PVS.* Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997), LNCS 1254. Springer, June 1997.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. *Abstractions from proofs.* Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2004), pages 232–244. ACM, January 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. *Lazy abstraction.* Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2002), pages 58–70. ACM, January 2002.

[JM09] Ranjit Jhala and Rupak Majumdar. *Software model checking.* ACM Computing Surveys (CSUR), volume 41, number 4, pages 1–54. ACM, October 2009.

[KW10] Daniel Kroening and Georg Weissenbacher. *Verification and Falsification of Programs with Loops using Predicate Abstraction.* Formal Aspects of Computing, volume 22, number 2, pages 105–124. Springer, March 2010.

[LNO06] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. *SMT Techniques for Fast Predicate Abstraction.* Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006), LNCS 4144, pages 424–437. Springer, August 2006.

[McM06] Kenneth L. McMillan. *Lazy Abstraction with Interpolants.* Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006), LNCS 4144, pages 123–136. Springer, August 2006.

[Neu09] Peter G. Neumann. *Risks to the public.* ACM SIGSOFT Software Engineering Notes, volume 34, number 5, pages 18–24. ACM, September 2009.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs.* Proceedings of the 11th International Conference on Compiler Construction (CC 2002), LNCS 2304, pages 213–228. Springer, April 2002.

[Seb07] Roberto Sebastiani. *Lazy Satisability Modulo Theories.* Journal on Satisfiability, Boolean Modeling and Computation, volume 3, number 3-4, pages 141–224. December 2007.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 26. März 2010

Philipp Wendler