Predicate Abstraction with Adjustable-Block Encoding

Dirk Beyer
Simon Fraser University / University of Passau

M. Erkan Keremoglu
Simon Fraser University, B.C., Canada

Philipp Wendler
University of Passau, Germany

***Abstract*—Several successful software model checkers are based on a technique called single-block encoding (SBE), which computes costly predicate abstractions after every single program operation. Large-block encoding (LBE) computes abstractions only after a large number of operations, and it was shown that this significantly improves the verification performance. In this work, we present adjustable-block encoding (ABE), a unifying framework that allows to express both previous approaches. In addition, it provides the flexibility to specify any block size between SBE and LBE, and also beyond LBE, through the adjustment of one single parameter. Such a unification of different concepts makes it easier to understand the fundamental properties of the analysis, and makes the differences of the variants more explicit. We evaluate different configurations on example C programs, and identify one that is currently the best.**

I. Introduction

Software model checking has been proven successful for increasing the quality of computer programs [2]. Several fundamental concepts were invented in the last decade which made it possible to scale the technology from tiny examples to real programs, e.g., device drivers [4], and to significantly improve the analysis precision, compared to traditional data-flow analyses. Predicate abstraction was introduced as an appropriate abstract domain [17], counterexample-guided abstraction refinement (CEGAR) makes it possible to automatically learn new facts to track [12], lazy abstraction performs expensive refinements only on relevant program paths [19], and interpolation is a sucessful technique to identify a small number of predicates that suffice to eliminate imprecise paths [15], [18].

The software model checker BLAST is an example of a tool that implements all of the above-mentioned concepts [7]. Such a tool implementation performs a reachability analysis along the edges of the control-flow automaton (CFA). The program counter is explicitly represented, and the data state is symbolically represented using predicates. The intermediate results are stored in an abstract reachability graph (ARG). Abstract successor states are obtained by computing the predicate abstraction of the strongest postcondition for a program operation, which involves querying a theorem prover. This category of implementing predicate abstraction can be characterized as *single-block encoding* (SBE), because every single control-flow edge of the program is transformed into a formula that is used for computing the abstract successor state. For a more detailed illustration of the general SBE approach on a concrete example, we refer the reader to the overview article [7].

* This research was supported in part by the Canadian NSERC grant RGPIN 341819-07.

Recently, a new approach was introduced which encodes many CFA edges into one formula, for computing the abstract successor. This approach is called *large-block encoding* (LBE) [6], and transforms the original CFA into a new, summarized CFA in which every edge represents a large subgraph (of the original CFA) that is free of loops. Solvers for satisfiability modulo theories (SMT) had continuously improved their expressiveness and performance, but the SBE approach did not take advantage of this additional power. Therefore, it was time to explore LBE, where a large part of the computational burden of the reachability analysis is delegated to an SMT solver. The experiments showed that LBE not only has a much better performance, but even a better precision (because it is feasible to use boolean instead of cartesian predicate abstraction). However, LBE has two drawbacks: First, it operates on a modified CFA which makes combinations with other abstract domains that operate on single edges impossible. Second, LBE is just one particular choice for how much of the program is encoded in one block and this choice is hard-coded into the verifier and cannot be changed. Our work addresses the need to explore the large space of choices from SBE to LBE, and also beyond LBE.

This article contributes a new approach that is called *adjustable-block encoding* (ABE), which unifies SBE and LBE in one single formalism and fills the gap of missing configurations. This new formalism, together with the corresponding tool implementation, makes it possible to perform experiments which were not possible before, i.e., in which the block encoding is adjustable as a parameter. ABE works on the original CFA and constructs the formulas for large blocks *on-the-fly* during the analysis, and in parallel to other domains (product domains). The number of operations that are encoded in one formula per abstraction step is freely adjustable using a so called block-adjustment operator. By modifying this parameter, ABE can not only operate like SBE or LBE, but can also express configurations with block encodings between SBE and LBE, as well as block encodings larger than LBE.

In our predicate analysis with adjustable-block encoding, every abstract state has two formulas to store the abstract data state: an abstraction formula and a path formula. The successor computation can operate in two different modes, either in abstraction mode or in non-abstraction mode. In a first step (same for both modes) the strongest postcondition for the path formula of the predecessor and the program operation is (syntactically) constructed as formula. In non-abstraction mode, this formula is stored as the path formula in the new state, and the abstraction formula is just copied.

```
1  int main() {
2      int i = 0;
3      while (i < 2) {
4          i++;
5      }
6      if (i != 2) {
7          ERROR: return 1;
8      }
9  }
```

Fig. 1. Simple example program

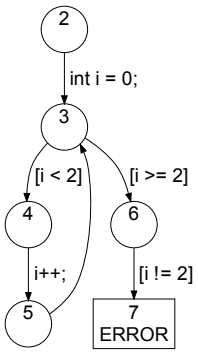

Fig. 2. Corresponding CFA

In abstraction mode, the boolean predicate abstraction of the formula is computed and stored as abstraction formula in the new state, and the path formula is set to *true*. At meet points in the control flow, and if the analysis is in non-abstraction mode, the path formulas of the two branches are combined via disjunction (resulting in a disjunctive path formula). In other words, as long as the analysis operates in non-abstraction mode, a disjunctive path formula is constructed that represents all program operations since the last abstraction formula was computed in abstraction mode. The mode is determined by the block-adjustment operator (analysis parameter).

Availability. Our experiments (implementation, benchmarks, logs) are available at `http://www.sosy-lab.org/~dbeyer/cpa-abe`. The archive includes an executable copy of the CPACHECKER system. For the complete system, cf. the CPACHECKER website.

Example. We illustrate ABE on the simple program in Fig. 1. Figure 2 shows the corresponding CFA (assume(p) is represented by [p]; we removed irrelevant parts from which the error location is not reachable). Nodes represent program locations and arrows represent program operations. We consider a predicate precision (the set of predicates that are tracked) that contains the predicates $i = 0$, $i = 1$, and $i = 2$. First we consider a block-adjustment operator that implements LBE on-the-fly, i.e., abstracting at loop heads and at the error location. The abstract reachability graph (ARG) is shown in Fig. 3. Nodes represent abstract states, and the numbers in the node are the CFA program location (top) and the unique state identifier (bottom). Nodes that are filled in grey represent abstraction states, and their abstraction formula is shown in the box attached to the abstraction state. Nodes with dashed circles represent abstract states that the analysis determines as unreachable (i.e., the result of the abstraction compuation is $false$). Such states are not added to the set of reachable states, therefore they do not have a unique state identifier. Note that the number of grey nodes shows exactly how many (costly) abstraction computations were necessary.

The analysis starts in non-abstraction mode, and is initialized with the formula *true* for both the abstraction formula ψ and the path formula φ. The analysis explores the path from location 2 to 3, creating abstract state $\frac{3}{2}$. Since location 3 is a loop head, state $\frac{3}{2}$ is an abstraction state and the computed abstraction formula is $i = 0$, the path formula φ is re-set to *true*. Locations 4 and 5 are no loop heads, so no abstraction

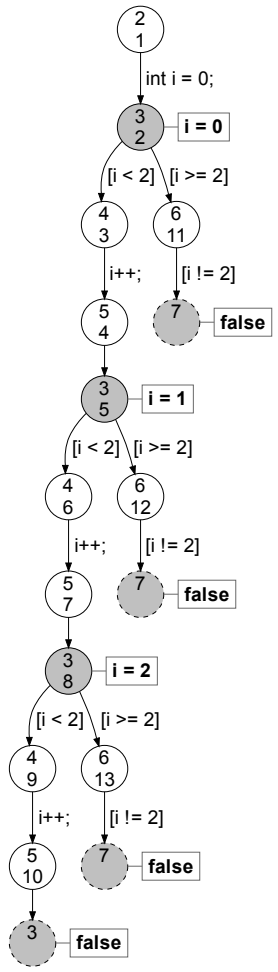

Fig. 3. ARG after analysis with large-block encoding (LBE)

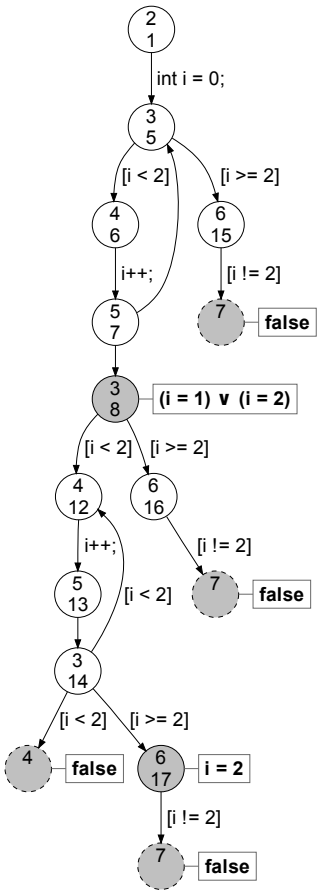

Fig. 4. ARG after analysis with blocks of length 7

is computed and instead only the path formula is extended by the operations on the edges to states $\frac{4}{3}$ and $\frac{5}{4}$. The abstraction formula is copied from their respective predecessor. When the analysis re-encounters location 3, an abstraction is computed again, this time with $i = 1$ as the result (state $\frac{3}{5}$). This process continues until the result of the abstraction computation is *false* (for the successor of $\frac{5}{10}$), which means that the new abstract state is not reachable and analysis can stop exploring this path. Note that $\frac{4}{9}$ and $\frac{5}{10}$ are already unreachable, but the analysis does not detect this, because the abstraction formula is not computed for such non-abstraction states. However, this does not cause a problem because all computations needed for the construction of non-abstraction states like $\frac{5}{10}$ are inexpensive compared with the cost of abstraction computations. The exploration of the remaining paths (those through location 6) is similar. At location 7, an abstraction is always computed because it is the error location, and thus the analysis checks the reachability of abstract states at this location. No such abstraction state is reachable, thus the program is safe.

Now we consider a block-adjustment operator that forces an abstraction computation if the longest path represented by the current path formula has length 7. The ARG is shown in Fig. 4. The analysis starts similarly to the previous example.

However, when it first encounters location 3 it does not compute an abstraction because the condition of the block-adjustment operator (length 7) is not yet fulfilled. Instead, it creates a non-abstraction state $\frac{3}{2}$ (which does not occur in the figure because it is later subsumed by the result of a merge). The same holds for $\frac{4}{3}$ and $\frac{5}{4}$. When the analysis reaches location 3 again, it creates state $\frac{3}{5}$ and immediately merges it with the existing state $\frac{3}{2}$, because both share the same location and the same abstraction formula (which is still the initial one). Therefore, abstract state $\frac{3}{2}$ is removed. The path formula of the new (merged) abstract state is the disjunction of the path formulas of both states, i.e., representing both the paths 2-3 and 2-3-4-5-3. The same happens at locations 4 and 6, creating states $\frac{4}{6}$ and $\frac{5}{7}$. But when the analysis encounters location 3 for the third time, the path formula represents the paths 2-3-4-5-3 and 2-3-4-5-3-4-5-3. The latter path contains 7 edges, thus an abstraction is computed. At this abstraction state, either the predicate $i = 1$ or the predicate $i = 2$ is true. Continuing, the analysis constructs the non-abstraction states $\frac{4}{9}$, $\frac{5}{10}$, $\frac{3}{11}$, $\frac{4}{12}$, $\frac{5}{13}$ and $\frac{3}{14}$. Again, the former three states are removed from the set of reached states because they are merged into the latter three states. All these six states are not merged with the previous states although some of them share a common program location, because the abstraction formula of the new states differs from the abstraction formula of the previous states. Also, abstraction states like $\frac{3}{8}$ are never changed by merge operations. The path formula of $\frac{3}{14}$ represents the paths 3-4-5-3 and 3-4-5-3-4-5-3. Thus, when the successors of this state are created, the length of the longest path represented by the path formula reaches 7 and an abstraction is computed. The successor at location 4 has the abstraction formula *false*, thus it is not added to the reached states. The abstraction formula of $\frac{6}{17}$ is $i = 2$. The analysis continues with the remaining paths, correctly determining that all paths leading to the error location are infeasible. Therefore the program is again reported as safe.

By choosing a good block-adjustment operator, the size of the blocks (the regions of the ARG that do not contain abstraction states) and the number of abstraction computations can be optimized. Larger blocks lead to fewer costly abstraction computations, but the problems given to the SMT solver are harder because the path formulas are more complex. With ABE, the reachable states do not necessarily form a tree, like for SBE and for LBE with preprocessing. However, note that the abstraction states still form a tree in both examples. In fact, this is true for all choices of the block-adjustment operator.

Related Work. Our work is based on the idea of stepwise exploring the reachable states of the program, using CEGAR to refine the abstraction, and symbolic techniques to operate on abstract data states. Existing example implementations of this category are SBE-based (SLAM [4] and BLAST [7]) or LBE-based [6]. The goal of our ABE-based approach is to make the configuration of the algorithm flexible, i.e., (1) to subsume the previous approaches (SBE, LBE) and (2) enable even larger encodings such that it is freely adjustable how much of the state-space exploration is done symbolically by the SMT solver. A different category of verification tools is based on the idea of performing a fully symbolic search. Examples are the model checker SATABS [14], which is based on CEGAR but operates fully symbolically, and the bounded model checker CBMC [13], which is targeted at finding bugs instead of proving safety. Fully symbolic search is also applied to large generated verification conditions, for example in the extended static checkers CALYSTO [1] and SPEC# [5]. The algorithm of McMillan is also based on the idea of lazy abstraction, but never performes predicate abstraction-based successor computations [21]. Our approach can be characterized as based on predicate abstraction [17], CEGAR [12], lazy abstraction [19], and interpolation [18].

II. Preliminaries

A. Programs and Control-Flow Automata

We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[1] We represent a program by a *control-flow automaton* (CFA). A CFA $A = (L, G)$ consists of a set L of program locations, which model the program counter l, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to another. The set of program variables that occur in operations from Ops is denoted by X. A *program* $P = (A, l_0, l_E)$ consists of a CFA $A = (L, G)$ (models the control flow of the program), an initial program location $l_0 \in L$ (models the program entry), and a target program location $l_E \in L$ (models the error loc.).

A *concrete data state* of a program is a variable assignment $c : X \to \mathbb{Z}$ that assigns to each variable an integer value. The set of all concrete data states of a program is denoted by $\mathscr{C}$. A set $r \subseteq \mathscr{C}$ of concrete data states is called *region*. We represent regions using first-order formulas (with free variables from X): a formula φ represents the set $[\![\varphi]\!]$ of all data states c that imply φ (i.e., $[\![\varphi]\!] = \{c \in \mathscr{C} \mid c \models \varphi\}$). A *concrete state* of a program is a pair (l, c), where $l \in L$ is a program location and c is a concrete data state. A pair (l, φ) represents the following set of concrete states: $\{(l, c) \mid c \models \varphi\}$. The *concrete semantics* of an operation $op \in Ops$ is defined by the strongest postcondition operator $\mathsf{SP}_{op}(\cdot)$: for a formula φ, $\mathsf{SP}_{op}(\varphi)$ represents the set of data states that are reachable from any of the states in the region represented by φ after the execution of op. Given a formula φ that represents a set of concrete data states, for an assignment operation $s := e$, we have $\mathsf{SP}_{\mathsf{s:=e}}(\varphi) = \exists \widehat{s} \colon \varphi_{[s \mapsto \widehat{s}]} \wedge (s = e_{[s \mapsto \widehat{s}]})$, and for an assume operation $assume(p)$, we have $\mathsf{SP}_{assume(\mathsf{p})}(\varphi) = \varphi \wedge p$.

A *path* σ is a sequence $\langle (op_1, l_1), ..., (op_n, l_n) \rangle$ of pairs of operations and locations. The path σ is called *program path* if σ starts with l_0 and for every i with $0 < i \leq n$ there exists a CFA edge $g = (l_{i-1}, op_i, l_i)$, i.e., σ represents a syntactical walk through the CFA. The *concrete semantics for a program*

[1] Our implementation CPACHECKER works on C programs that are given in CIL intermediate language [22]; non-recursive function calls are supported.

path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$ is defined as the successive application of the strongest postoperator for each operation: $\mathsf{SP}_\sigma(\varphi) = \mathsf{SP}_{op_n}(...\mathsf{SP}_{op_1}(\varphi)...)$. The formula $\mathsf{SP}_\sigma(\varphi)$ is called *path formula*. The set of concrete states that result from running σ is represented by the pair $(l_n, \mathsf{SP}_\sigma(true))$. A program path σ is *feasible* if $\mathsf{SP}_\sigma(true)$ is satisfiable. A concrete state (l_n, c_n) is called *reachable* if there exists a feasible program path σ whose final location is l_n and such that $c_n \models \mathsf{SP}_\sigma(true)$. A location l is reachable if there exists a concrete state c such that (l, c) is reachable. A program is *safe* if l_E is not reachable.

B. Predicate Precision and Boolean Predicate Abstraction

Let $\mathscr{P}$ be a set of predicates over program variables in a quantifier-free theory $\mathscr{T}$. A *formula* φ is a boolean combination of predicates from $\mathscr{P}$. A *precision for formulas* is a finite subset $\pi \subset \mathscr{P}$ of predicates. A *precision for programs* is a function $\Pi : L \to 2^{\mathscr{P}}$, which assigns to each program location a precision for formulas. The *boolean predicate abstraction* $(\varphi)^\pi$ *of a formula* φ is the strongest boolean combination of predicates from the precision π that is entailed by φ. Such a predicate abstraction of a formula φ, which represents a region of concrete program states, is used as an *abstract data state* (i.e., an abstract representation of the region) in program verification. For a formula φ and a precision π, the boolean predicate abstraction $(\varphi)^\pi$ of φ can be computed by querying an SMT solver in the following way: For each predicate $p_i \in \pi$, we introduce a propositional variable v_i. Now we ask the solver to enumerate all satisfying assignments of $v_1, ..., v_{|\pi|}$ in the formula $\varphi \wedge \bigwedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$. For each satisfying assignment, we construct a conjunction of all predicates from π whose corresponding propositional variable occurs positive in the assignment. The disjunction of all such conjunctions is the boolean predicate abstraction for φ. An abstract strongest postoperator for a predicate abstraction with precision π and a program operation *op*, which transforms an abstract data state φ into its successor φ', can be defined by applying first the strongest postcondition operator and then the predicate abstraction, i.e., $\varphi' = (\mathsf{SP}_{op}(\varphi))^\pi$. For more details, we refer the reader to the work of Ball et al. and Lahiri et al. [3], [20].

III. Adjustable-Block Encoding

In ABE, the predicate abstraction is not computed after every CFA edge, but only at certain abstract states, which we call *abstraction states* (the other abstract states are called non-abstraction states). On paths between two abstraction computations, the strongest postcondition of the path(s) is stored in a second formula of the abstract state, which we call *disjunctive path formula*. Therefore, every abstract state of ABE contains two formulas ψ and φ, where the *abstraction formula* ψ is the result of an abstraction computation and the disjunctive path formula φ represents the strongest postcondition since the last abstraction state was computed. Given a CFA edge $g = (l, op, l')$ and an abstract state with ψ and φ, the abstract successor either extends the path formula φ only (which is a purely syntactical operation), or computes a new abstraction formula ψ and resets φ. Where to compute abstractions (and thus the block size) is determined by the so-called block-adjustment operator blk as follows: If $\mathsf{blk}(e, g)$ returns *false* (no abstraction computation, i.e., the abstract state e is a non-abstraction state), the abstract successor contains ψ (unchanged) and $\mathsf{SP}_{op}(\varphi)$ (as the new φ). If $\mathsf{blk}(e, g)$ returns *true* (e is abstraction state), the abstract successor contains the formula that results from the abstraction of $\psi \wedge \varphi$ as the new abstraction formula and *true* as the new disjunctive path formula. If $\psi \wedge \varphi$ is unsatisfiable for an abstract state e, then e is not reachable.

A. CPA for Adjustable-Block Encoding

We formalize adjustable-block encoding (ABE) as a configurable program analysis (CPA) [8]. This allows us to use the flexibility of the CPA operators to describe how the analysis operates without changing the general iteration algorithm (cf. Alg. *CPA*). The configurable program analysis for adjustable-block encoding $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a stop operator stop, which are defined as follows. (Given a program $P = (A, l_0, l_E)$, we use X for denoting the set of program variables occuring in P, $\mathscr{P}$ for the set of quantifier-free predicates over variables from X, and $\Pi : L \to 2^{\mathscr{P}}$ for the precision of the predicate abstraction.)

1. The **abstract domain** $D = (C, \mathscr{E}, [\![\cdot]\!])$ is a tuple that consists of a set C of concrete states, a semi-lattice $\mathscr{E} = (E, \top, \sqsubseteq, \sqcup)$, and a concretization function $[\![\cdot]\!] : E \to C$. The lattice elements $e \in E$ are also called abstract states, and are tuples $(l, \psi, l^\psi, \varphi) \in (L \cup \{l_\top\}) \times \mathscr{P} \times (L \cup \{l_\top\}) \times \mathscr{P}$, where l models the program counter, the abstraction formula ψ is a boolean combination of predicates that occur in Π, l^ψ is the location at which ψ was computed, and φ is a disjunctive path formula representing some or all paths from l^ψ to l. Note that an abstraction state has always $l = l^\psi$ and $\varphi = true$. The top element of the lattice is the abstract state $\top = (l_\top, true, l_\top, true)$. 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:

$$e_1 \sqsubseteq e_2 \iff (e_2 = \top) \vee ((l_1 = l_2) \wedge (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2))$$

The join operator $\sqcup : E \times E \to E$ yields the least upper bound of the two operands, according to the partial order.

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

$$\begin{cases} (\varphi' = true) \wedge \left(\psi' = (\mathsf{SP}_{op}(\varphi \wedge \psi))^{\Pi(l')}\right) \wedge (l^{\psi\prime} = l') \\ \qquad\qquad\qquad\qquad \text{if } \mathsf{blk}(e, g) \vee (l' = l_E) \\ (\varphi' = \mathsf{SP}_{op}(\varphi)) \wedge (\psi' = \psi) \wedge (l^{\psi\prime} = l^\psi) \qquad \text{otherwise} \end{cases}$$

The 'mode' of the transfer relation, i.e., when to compute abstractions, is determined by a block-adjustment operator

Algorithm 1 $CPA(\mathbb{D}, e_0)$ (taken from [8])

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

$\mathsf{blk} : E \times G \to \mathbb{B}$, which maps an abstract state e and a CFA edge g to *true* or *false*. The operator blk is given as parameter to the analysis. The second case does not compute an abstraction, but purely syntactically assembles the precise strongest postcondition [2]. Thus, the choice of blk determines the block-encoding (i.e., how much to collect in the path formula before abstraction). Most instances of the block-adjustment operator will eventually return *true* for every path through the CFA, otherwise the analysis might not terminate if the program contains loops. The precision of the predicate abstraction can vary between program locations (parsimonious precision [7]).

3. The **merge operator** $\mathsf{merge} : E \times E \to E$ for two abstract states $e_1 = (l_1, \psi_1, {l^\psi}_1, \varphi_1)$ and $e_2 = (l_2, \psi_2, {l^\psi}_2, \varphi_2)$ is defined as follows: $\mathsf{merge}(e_1, e_2) =$

$$\begin{cases} (l_2, \psi_2, {l^\psi}_2, \varphi_1 \vee \varphi_2) & \text{if } (l_1 = l_2) \wedge (\psi_1 = \psi_2) \wedge ({l^\psi}_1 = {l^\psi}_2) \\ e_2 & \text{otherwise} \end{cases}$$

This operator combines the two abstract states using a disjunctive path formula, if the location of the abstract states is the same and they were derived from the same abstraction states, i.e., the abstraction formulas are equal and were computed at the same program location [3].

4. The **stop operator** $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ checks if e is covered by another state in the reached set:

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

[2] The strongest postcondition as defined in the preliminaries in fact contains existential quantifiers. However, our implementation of the transfer relation uses an SP that operates on SSA-like quantifier-free formulas.

[3] Two identical abstraction states never exist in the reached set due to the stop operator, which would eliminate the second instance of the same abstraction state before insertion into the reached set. Thus, the ARG restricted to abstraction states still represents a tree (ART).

B. Discussion: SBE, LBE, BMC, and in Between

A fundamental improvement of adjustable-block encoding over the previous work with blocks hard-coded in the pre-processed CFA is that now other abstract domains that the ABE analysis is combined with, can work each with different (perhaps also adjustable) block sizes, which are not dictated by the pre-processed CFA. The great flexibility of ABE results from the possibility to freely choose the blk operator. Two particular possibilities are blk^{sbe} and blk^{lbe}. The operator blk^{sbe} returns always *true*, and thus the transfer computes the predicate abstraction after every CFA edge. The operator blk^{lbe} returns *true* if the successor location of the given edge is the head location of a loop, and thus the transfer computes the predicate abstraction at loop heads. Therefore, the analysis can easily be configured to behave exactly like SBE or LBE. Another possible choice for blk is 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. But the decision made by blk does not necessarily have to be based only on statically available information. We could for example measure the memory consumption of the path formula and compute abstractions if the path formulas become too large. Or we could measure the time needed to compute the abstractions and adjust the block encoding such that a single computation does not take more than a certain amount of time. Thus, one could write a block-adjustment operator that is tailored to the SMT solver that is used, i.e., to delegate problems to the solver that are large enough to benefit from the SMT technology, and at the same time small enough to not overwhelm the solver. In our experiments, we demostrate the usefulness of the ABE approach using a few simple choices for the operator blk. In particular, the experiments indicate that useful block-adjustment operators should respect the control-flow structure of the program.

We did not describe how the precision Π for programs is computed, because we use a standard approach that is based on CEGAR, lazy abstraction, and Craig interpolation. We consider only abstraction states for the abstract reachability tree (ART). The merge operator ensures that the abstraction states form a tree (abstraction states are never changed by merge). The formulas of the error path are the (disjunctive) path formulas that were constructed during the creation of the abstraction states along this path and which were used as inputs for the abstraction computation. The only difference is that now a single formula represents one or several paths between two arbitrary locations of the CFA, and not necessarily only one CFA edge as before. The interpolation will then produce predicates for those locations at which an abstraction was computed, so the new predicates will be used in the next iteration of the analysis if the block-adjustment operator returns the same value. The possiblity of dynamic block-adjustment operators, i.e., determining different abstract states as abstraction states depending on the overall progress of the analysis, raises the interesting question of where to add the predicates extracted from the interpolants. Currently, we refine

the predicate precision of the program only at the abstraction states, but we could in principle also add the predicates to all locations between the previous and current abstraction state.

IV. Experiments

Implementation. We implemented adjustable-block encoding in CPACHECKER, which is a software-verification framework based on configurable program analysis. The tool accepts programs in C Intermediate Language [22] (other C programs can be pre-processed with the tool CIL). CPACHECKER uses MATHSAT [11] and CSISAT [9] as SMT solvers.

Our implementation uses the following optimizations: (1) The feasibility of abstract paths is checked only at abstraction states. This does not negatively affect the precision of the analysis because abstract states with the error location are always abstraction states. (2) Instead of constructing postconditions that include existential quantifiers, we leave the variables in the path formulas and use a simple form of skolemization, which is equivalent to static single-assignment (SSA) form [16] (well-known from compilers). A tutorial-like example of this process is provided in an article about BLAST [7]. (3) When the operator stop checks if an abstract state e is covered by a non-abstraction state e', we do not perform a full SMT check for the implication that the partial order requires; instead we do a quick syntactical check that compares the path formulas of both states. This check will correctly detect that e is covered by e' if e was merged into e', but may fail in other situations. This is sound, and faster than a full SMT check.

Benchmark Programs. We experimented with three groups of C programs, which are similar to those previously used [6]. The first group (`test_locks_*`) was artificially created to show that SBE leads to exponentially many abstract states. Several nested locks are acquired and released in a loop. The number in the name indicates the number of locks in the program. The second group contains several (parts of) drivers from the Windows NT kernel. The third group (`s3_*`) was taken from the SSH suite. The code contains a simplified version of the state machine handling the communication according to the SSH protocol. Both the NT drivers and the SSH examples were pre-processed manually in order to remove heap accesses, and automatically with CIL v1.3.6. The examples with `BUG` in the name have artificially inserted bugs that cause assertions to fail. All examples are included in the CPACHECKER repository together with the used configurations.

All experiments were performed on a machine with 2.8 GHz and 4 GB of RAM. The operating system was Ubuntu 9.10 (64 bit), using Linux 2.6.31 as kernel and OpenJDK 1.6 as Java virtual machine. We used CPACHECKER, branch 'abe', revision 1457, with MATHSAT 4.28 as SMT solver. The times are reported in seconds and rounded to three significant digits. In cases where CPACHECKER needed either more than 1800 s or more than 4 GB of RAM, the analysis was aborted, indicated by "> 1800" or "MO", respectively. CPACHECKER reports the correct verification result in all cases, i.e., a counterexample for all programs with `BUG`, and safety for all other programs.

Configurations. We experimented with different choices of the block-adjustment operator of ABE, which we classify into three categories: (1) we repeat the experiments with LBE from previous work [6], (2) we perform new experiments to explore the spectrum of encodings between SBE and LBE, and (3) we explore new encodings larger than LBE. Our implementation supports functions, and thus we extend the operator blk^{lbe} such that it returns *true* at loop heads *and* function entries/returns. We measure the length of a block that is encoded in an abstract state e as the length (in ops) of the longest path represented in the disjuncive path formula of e.

We have also experimented with cartesian vs. boolean abstraction, and not only re-confirm the results from previous experiments [6] (SBE: cartesian works best, LBE: boolean is best); we conclude that cartesian abstraction becomes unusably imprecise as soon as the block length is more than 1 op. Thus, boolean abstraction must be used for all non-SBE encodings.

A. LBE: Pre-Processed versus On-the-Fly

Due to the overhead that is caused by the on-the-fly encoding and some additional abstraction computations, a certain performance loss is expected. We started our experiments with confirming that the negative impact of the overhead on the performance is not dramatic. The results are reported in

Program	Pre-proc. LBE	LBE (blk^{lbe})
test_locks_5.c	.170	.483
test_locks_6.c	.370	.398
test_locks_7.c	.237	.875
test_locks_8.c	.305	.437
test_locks_9.c	.202	.510
test_locks_10.c	.266	.746
test_locks_11.c	.256	.416
test_locks_12.c	.248	.486
test_locks_13.c	.240	.769
test_locks_14.c	.227	.787
test_locks_15.c	.466	.896
cdaudio1.sim.c	11.7	51.5
diskperf1.sim.c	537	146
floppy3.sim.c	7.04	20.1
floppy4.sim.c	8.35	32.2
kbfiltr1.sim.c	1.27	2.57
kbfiltr2.sim.c	1.73	3.75
cdaudio1_BUG.sim.c	5.26	32.5
floppy3_BUG.sim.c	2.97	11.1
floppy4_BUG.sim.c	4.58	20.1
kbfiltr2_BUG.sim.c	1.96	2.28
s3_clnt_1.sim.c	15.9	14.6
s3_clnt_2.sim.c	12.8	35.4
s3_clnt_3.sim.c	19.5	17.8
s3_clnt_4.sim.c	36.6	9.59
s3_srvr_1.sim.c	16.6	31.2
s3_srvr_2.sim.c	107	86.7
s3_srvr_3.sim.c	109	14.1
s3_srvr_4.sim.c	441	160
s3_srvr_6.sim.c	456	45.7
s3_srvr_7.sim.c	321	136
s3_srvr_8.sim.c	>1800	21.2
s3_clnt_1_BUG.sim.c	1.22	2.81
s3_clnt_2_BUG.sim.c	2.12	2.06
s3_clnt_3_BUG.sim.c	1.26	3.14
s3_clnt_4_BUG.sim.c	2.03	2.54
s3_srvr_1_BUG.sim.c	1.43	1.62
s3_srvr_2_BUG.sim.c	1.55	2.71

TABLE I
COMPARISON OF PRE-PROCESSED LBE WITH ADJUSTED LBE (blk^{lbe})

Program	SBE	$k=10$	$k=20$	$k=30$	$k=40$	$k=50$	$k=60$	$k=70$	$k=80$	$k=90$	$k=100$	LBE
test_locks_5.c	6.36	3.42	1.02	1.29	.367	.695	.397	.292	.587	.468	.507	.483
test_locks_6.c	13.1	3.03	1.90	1.36	.690	.334	.527	.428	.637	.790	.323	.398
test_locks_7.c	34.8	5.71	1.30	3.26	.516	1.06	.800	.326	.591	.355	.807	.875
test_locks_8.c	102	25.8	3.82	1.86	1.20	1.27	.414	.392	.670	.575	.680	.437
test_locks_9.c	298	67.7	12.4	6.97	1.67	1.63	.543	.454	.551	.667	.705	.510
test_locks_10.c	1250	109	7.53	6.59	3.79	1.24	.679	.588	.805	.845	.993	.746
test_locks_11.c	>1800	244	26.7	4.71	6.73	1.71	.906	.992	1.20	.905	.418	.416
test_locks_12.c	>1800	>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	>1800	MO	>1800	580	19.6	17.6	4.61	2.25	2.07	1.13	.915	.787
test_locks_15.c	>1800	>1800	>1800	>1800	32.2	22.3	23.1	5.56	2.71	2.46	1.40	.896
cdaudio1.sim.c	MO	210	119	51.9	52.9	54.5	49.0	52.6	58.1	53.8	53.5	51.5
diskperf1.sim.c	MO	855	155	171	163	168	158	152	154	146	167	146
floppy3.sim.c	559	80.5	23.6	19.3	25.5	23.1	20.0	21.0	21.1	19.8	17.9	20.1
floppy4.sim.c	MO	212	54.0	28.8	41.4	39.2	35.2	31.7	32.6	32.8	44.6	32.2
kbfiltr1.sim.c	48.2	10.1	3.72	2.66	3.28	2.82	2.15	2.49	1.83	1.89	2.81	2.57
kbfiltr2.sim.c	128	59.1	10.2	5.26	5.90	7.93	4.12	4.56	4.19	4.67	3.94	3.75
cdaudio1_BUG.sim.c	158	106	151	32.6	36.7	38.6	32.7	35.5	40.0	33.4	31.9	32.5
floppy3_BUG.sim.c	75.8	45.9	13.9	12.1	13.9	11.3	11.2	9.38	9.11	10.5	10.4	11.1
floppy4_BUG.sim.c	77.4	150	39.0	16.9	30.4	31.3	26.1	21.3	22.2	23.3	23.1	20.1
kbfiltr2_BUG.sim.c	156	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.sim.c	MO	MO	MO	MO	MO	41.3	27.8	13.4	10.8	445	45.0	14.6
s3_clnt_2.sim.c	MO	MO	MO	MO	MO	34.4	45.0	16.2	12.8	569	49.1	35.4
s3_clnt_3.sim.c	MO	MO	MO	MO	MO	45.7	238	309	24.6	MO	36.4	17.8
s3_clnt_4.sim.c	MO	MO	MO	MO	MO	38.1	17.7	24.2	9.53	441	28.4	9.59
s3_srvr_1.sim.c	>1800	MO	MO	MO	MO	43.7	MO	712	113	MO	47.8	31.2
s3_srvr_2.sim.c	MO	MO	MO	MO	MO	462	33.2	MO	340	MO	98.5	86.7
s3_srvr_3.sim.c	>1800	MO	MO	MO	MO	32.9	11.5	31.1	24.7	MO	MO	14.1
s3_srvr_4.sim.c	>1800	MO	MO	MO	MO	325	56.4	12.1	22.0	MO	45.6	160
s3_srvr_6.sim.c	>1800	MO	MO	MO	MO	MO	83.8	638	MO	50.8	MO	45.7
s3_srvr_7.sim.c	>1800	MO	MO	MO	MO	MO	133	458	MO	MO	315	136
s3_srvr_8.sim.c	>1800	MO	MO	MO	MO	MO	18.9	42.7	26.8	155	565	21.2
s3_clnt_1_BUG.sim.c	667	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_BUG.sim.c	677	135	26.6	14.2	10.2	3.93	3.05	1.94	2.59	6.47	3.58	2.06
s3_clnt_3_BUG.sim.c	653	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_BUG.sim.c	646	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_BUG.sim.c	42.2	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_BUG.sim.c	35.6	60.4	6.18	2.72	4.80	2.65	1.09	2.03	1.61	5.00	3.32	2.71

TABLE II
RESULTS FOR blk^{sbe}, FOR blk_k^{lbe} WITH k FROM 10 TO 100, AND FOR blk^{lbe}

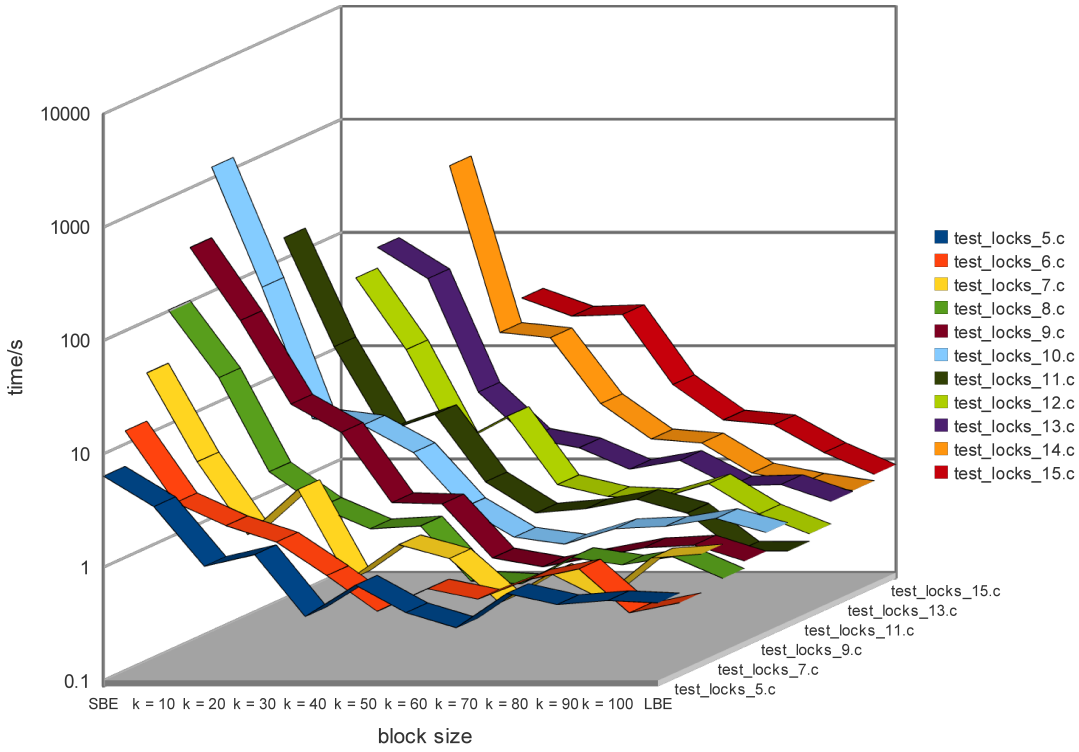

Fig. 5. Results for blk^{sbe}, for blk_k^{lbe} with k from 10 to 100, and for blk^{lbe}

Program	LBE		$k=50$		$k=100$		$k=150$		$k=200$		$k=250$		$k=300$	
cdaudio1_BUG.sim.c	32.5	26989	451	49240	168	6428	11.1	444	12.9	114	15.9	105	13.4	181
floppy3_BUG.sim.c	11.1	4059	51.4	4697	20.1	1207	6.17	217	7.81	99	10.9	112	4.03	43
floppy4_BUG.sim.c	20.1	11066	87.7	7734	43.7	2491	11.8	177	14.2	167	10.4	161	6.63	55
kbfiltr2_BUG.sim.c	2.28	660	3.75	424	1.94	57	1.45	52	2.22	24	1.07	0	1.10	0
s3_clnt_1_BUG.sim.c	2.81	16	14.6	743	MO	-	5.39	38	3.63	33	4.67	10	6.57	20
s3_clnt_2_BUG.sim.c	2.06	20	3.70	146	2.94	51	3.41	28	MO	-	4.48	9	5.07	18
s3_clnt_3_BUG.sim.c	3.14	22	14.6	795	MO	-	MO	-	13.1	44	7.06	7	5.27	7
s3_clnt_4_BUG.sim.c	2.54	22	5.21	218	4.46	78	MO	-	13.2	125	5.32	9	6.17	19
s3_srvr_1_BUG.sim.c	1.62	13	6.01	303	1.28	38	3.24	10	MO	-	1.90	2	2.13	7
s3_srvr_2_BUG.sim.c	2.71	13	4.61	306	MO	-	2.24	10	MO	-	1.97	2	2.01	7
test_locks_5.c	.483	4	1.21	22	1.03	10	2.05	7	1.09	5	1.65	3	2.49	2
test_locks_6.c	.398	4	71.4	2258	2.26	84	2.22	40	2.18	12	1.11	1	1.47	1
test_locks_7.c	.875	4	3.17	120	2.48	17	1.11	1	1.66	10	7.16	26	5.90	60
test_locks_8.c	.437	4	3.33	180	.578	1	1.16	14	8.71	144	1.78	1	5.49	7
test_locks_9.c	.510	4	2.37	89	.880	2	11.2	192	3.49	19	2.62	10	15.5	47
test_locks_10.c	.746	4	1.83	118	.677	10	4.97	144	32.1	114	11.2	38	1.57	1
test_locks_11.c	.416	4	3.85	164	1.59	22	3.80	27	1.75	18	8.51	95	3.19	14
test_locks_12.c	.486	4	33.0	1985	MO	-	1.12	1	17.2	98	1.29	1	185	537
test_locks_13.c	.769	4	24.4	285	324	4628	.626	1	37.8	470	1.48	3	24.1	232
test_locks_14.c	.787	4	179	79	77.1	1202	1.01	2	10.0	220	4.43	22	27.8	107
test_locks_15.c	.896	4	1580	1268	154	2234	1.80	10	4.93	37	367	1255	1.28	1

TABLE III
RESULTS AND NUMBER OF ABSTRACTIONS FOR blk^{lbe} AND FOR blk_k WITH k FROM 50 TO 300

Table I. Column 'Pre-proc. LBE' reports the performance of the previous implementation [6], which first transforms the CFAs of the program in a pre-processing step into new CFAs that reflect the large-block encoding, and then the analysis is performed. Column 'LBE (blk^{lbe})' reports the performance of the new, more flexible implementation, with the block operator adjusted to blk^{lbe}. The simple old implementation is faster on most example programs, as expected, but the difference is not dramatic, and also inconsistent, i.e., there are several examples on which the new approach performs as good or even better. Thus, although it might seem wasteful to explore a huge number of extra states without performing any abstraction, just to assemble the strongest-postcondition formula for the encoded block on-the-fly, this table shows that in fact the overhead is not dramatic.

B. Block Sizes between SBE and LBE

The second set, of novel configurations, is based on the new block-adjustment operator $\mathsf{blk}_k^{lbe} : E \times G \to \mathbb{B}$, which is defined as the disjunction $\mathsf{blk}^{lbe} \vee \mathsf{blk}_k$. The operator blk_k^{lbe} returns *true* if either the longest path represented by the disjunctive path formula of the abstract state is longer than $k \in \mathbb{N} \setminus \{0\}$ or the successor location is a loop/function head. Only a few examples have blocks longer than 100 ops when analyzed with LBE, thus, we analyze the examples for $k \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.

Table II shows the results for the three groups of example programs. The examples `test_locks_*` and the NT drivers show an exponential performance improvement with growing blocks. The diagram with a logarithmic time axis in Fig. 5 illustrates this for the first group of examples. The blocks of blk_k^{lbe} are never longer than in LBE, and thus, there is no further performance improvement beyond a certain program-dependent threshold. The SSH programs with artificial bugs follow the same trend. The safe SSH programs do not follow such a clear trend and instead, their performance extremely depends on the block size. This indicates the superiority of blk^{lbe} over blk_k^{lbe} for blocks that are smaller than in the LBE encoding: the operator blk_k cuts off the formulas at an arbitrary position, ignoring the structure of the control flow and thus destroying the advantage of encoding larger blocks. LBE, i.e., the encoding with the largest blocks, is the only configuration that can verify all examples, and is the best for most examples.

It is important that the block-encoding encloses 'whole structures', i.e., that a block not only contains a few branches, but that it actually contains the branches until they meet again (for example, a whole 'if' structure). This is demonstrated by the fact that encodings smaller than LBE are sometimes not performing well (cf. `s3_clnt_3.c` with $k = 50$ vs. $k = 60$; in particular, note the MO for $k = 90$). This is particularly important for loop unrollings: coverage checks can be done most efficiently at abstraction states, thus, abstraction computations should ideally occur at matching locations of the loop body.

C. Block Sizes larger than LBE

In the third set of experiments, we evaluated new configurations with blocks larger than LBE. It is known that shallow bugs can efficiently be found by a technique called bounded model checking, where programs are unrolled up to a given bound of the length, and a formula is constructed which is satisfiable iff one of the modeled program paths reaches the error location. We apply a similar technique to find bugs using our ABE approach: in this set of experiments, we use the block-adjustment operator $\mathsf{blk}_k : E \times G \to \mathbb{B}$, which returns *true* if the block is k operations long, with $k \in \{50, 100, 150, 200, 250, 300\}$.

Table III reports the time and number of abstraction computations needed to find the error, in the first part of the table. The benefit of ever larger block encodings is clearly indicated. The performance of this configuration for finding bugs is almost comparable to the performance of a highly tuned tool for bounded model checking (BMC) [10]: we analyzed the programs with CBMC [13] and the runtimes were less than 6 s for every NT driver example with bug. It is interesting to consider the number of abstractions in our table; there are even two cases where the large size of the block encoding makes it possible to find the bug without any abstraction computation or refinement step (this would be equivalent to BMC).

As the results look promising, very large block encodings might be a way to reduce the number of abstraction computations, which in turn improves both precision *and* performance. Therefore, we also experimented with the examples that do not have bugs. The performance of the examples `test_locks_*` are shown in the second part of Table III. The results show that the performance can dramatically decrease if the block is terminated after a certain number of operations regardless of the control-flow structure. The performance of the NT driver examples without bug did not improve, because the effect of the loop bodies seems efficiently represented by the abstractions at the loop heads (or loop bodies are not relevant for the property to verify). The results for the SSH programs were mixed. Only some configurations provide better performance than LBE for a few programs. However, there were many examples for which much more time is needed, or the analysis even fails to terminate. Almost all time is spent by the SMT solver while computing Craig interpolants, and the resulting formulas are sometimes huge. The SMT solvers seem to be overwhelmed by the complexity of the large disjunctive path formulas. A comparison of different SMT solvers (MATHSAT, CSISAT) shows that different solvers perform well on different examples. Thus, we can conclude that there is much room for improvement when a new generation of SMT solvers is available which can handle large interpolation queries (only three interpolating SMT solvers are currently available: MATHSAT [11], CSISAT [9], FOCI [21]).

V. Conclusion

Software model checking largely depends on automated theorem proving, and the efficiency and precision have significantly improved over the last years due to ever better theorem provers. We have designed and implemented a model-checking approach which makes it possible to flexibly choose how much of the state-space exploration is delegated to a theorem prover. A previous project had already indicated that it is highly beneficial to design the model-checking process such that *larger* queries can be given to a theorem prover, and less state-space is explored by the software model checker itself [6]. Our work provides answers to several new experimental questions: (1) We should generally use boolean abstraction in software model checking, because cartesian abstraction is feasible only for one (the traditional, SBE) configuration. (2) On the full spectrum between single-block encoding (SBE) and large-block encoding (LBE), there is no configuration of the block size that is generally better than LBE. (3) Encodings in the spectrum far beyond LBE can significantly improve the performance for finding bugs, similar to bounded model checking. We have also identified room for improvement for block encodings larger than LBE. [4] We leave it for future work to explore improvements of interpolation procedures, and to assemble structurally better encoding formulas that are 'easier' for theorem provers, restrict the size of the interpolation queries, or help the SMT solver where needed by keeping structures explicit. We found it convenient to formalize our concept of ABE using the framework of configurable program analysis [8]; but we have only specified explicitly what ABE means for a predicate-analysis domain, and have not yet designed any other abstract domains (e.g., numerical domains) with adjustable-block encoding.

[4] For example, the "very large block" encodings should not be defined as a strict k-bound, but respect the control structure of the program (e.g., compute an abstraction after each control-flow subgraph of size more than k) — the ABE approach opens a large spectrum of possibilities.

References

[1] D. Babic and A. J. Hu, "CALYSTO: Scalable and precise extended static checking," in *Proc. ICSE*. ACM, 2008, pp. 211–220.

[2] T. Ball, B. Cook, V. Levin, and S. Rajamani, "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft," in *Proc. IFM*, LNCS 2999. Springer, 2004, pp. 1–20.

[3] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and cartesian abstractions for model checking C programs," in *Proc. TACAS*, LNCS 2031. Springer, 2001, pp. 268–283.

[4] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proc. POPL*. ACM, 2002, pp. 1–3.

[5] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," in *Proc. PASTE*. ACM, 2005, pp. 82–87.

[6] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Proc. FMCAD*. IEEE, 2009, pp. 25–32.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST: Applications to software engineering," *Int. J. Softw. Tools Technol. Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[8] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Proc. CAV*, LNCS 4590. Springer, 2007, pp. 504–518.

[9] D. Beyer, D. Zufferey, and R. Majumdar, "CSISAT: Interpolation for LA+EUF," in *Proc. CAV*, LNCS 5123. Springer, 2008, pp. 304–308.

[10] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS*, LNCS 1579. Springer, 1999, pp. 193–207.

[11] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MATHSAT 4 SMT solver," in *Proc. CAV*, LNCS 5123. Springer, 2008, pp. 299–303.

[12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[13] E. M. Clarke, D. Kröning, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. TACAS*, LNCS 2988. Springer, 2004, pp. 168–176.

[14] E. M. Clarke, D. Kröning, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Proc. TACAS*, LNCS 3440. Springer, 2005, pp. 570–574.

[15] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek, "Efficiently computing static single-assignment form and the program dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[17] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Proc. CAV*, LNCS 1254. Springer, 1997, pp. 72–83.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *Proc. POPL*. ACM, 2004, pp. 232–244.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proc. POPL*. ACM, 2002, pp. 58–70.

[20] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT techniques for fast predicate abstraction," in *Proc. CAV*, LNCS 4144. Springer, 2006, pp. 424–437.

[21] K. L. McMillan, "Lazy abstraction with interpolants," in *Proc. CAV*, LNCS 4144. Springer, 2006, pp. 123–136.

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. CC*, LNCS 2304. Springer, 2002, pp. 213–228.