UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL’INFORMAZIONE

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.disi.unitn.it

SOFTWARE MODEL CHECKING VIA LARGE-BLOCK ENCODING

Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu and Roberto Sebastiani

May 2009

Technical Report # DISI-09-026

Software Model Checking via Large-Block Encoding [1]

Dirk Beyer [2] Alessandro Cimatti [3] Alberto Griggio [2,4] M. Erkan Keremoglu [2] Roberto Sebastiani [4]
Simon Fraser Univ. FBK-irst Univ. of Trento & Simon Fraser Univ. Simon Fraser Univ. Univ. of Trento

Abstract—**The construction and analysis of an abstract reachability tree (ART) are the basis for a successful method for software verification. The ART represents unwindings of the control-flow graph of the program. Traditionally, a transition of the ART represents a single block of the program, and therefore, we call this approach single-block encoding (SBE). SBE may result in a huge number of program paths to be explored, which constitutes a fundamental source of inefficiency. We propose a generalization of the approach, in which transitions of the ART represent larger portions of the program; we call this approach large-block encoding (LBE). LBE may reduce the number of paths to be explored up to exponentially. Within this framework, we also investigate symbolic representations: for representing abstract states, in addition to conjunctions as used in SBE, we investigate the use of arbitrary Boolean formulas; for computing abstract-successor states, in addition to Cartesian predicate abstraction as used in SBE, we investigate the use of Boolean predicate abstraction. The new encoding leverages the efficiency of state-of-the-art SMT solvers, which can symbolically compute abstract large-block successors. Our experiments on benchmark C programs show that the large-block encoding outperforms the single-block encoding.**

I. Introduction

Software model checking is an effective technique for software verification. Several advances in the field have lead to tools that are able to verify programs of considerable size, and show significant advantages over traditional techniques in terms of precision of the analysis (e.g., SLAM [3] and BLAST [4]). However, efficiency and scalability remain major concerns in software model checking and hamper the adaptation of the techniques in industrial practice. A successful approach to software model checking is based on the construction and analysis of an abstract reachability tree (ART), and predicate abstraction is one of the favorite abstract domains. The ART represents unwindings of the control-flow graph of the program. The search is usually guided by the control flow of the program. Nodes of the ART typically consist of the control-flow location, the call stack, and formulas that represent the data states. During the refinement process, the ART nodes are incrementally refined.

In the traditional ART approach, each program operation (assignment operation, assume operation, function call, function return) is represented by a single edge in the ART. Therefore, we call this approach *single-block encoding* (SBE). A fundamental source of inefficiency of this approach is the fact that the control-flow of the program can induce a huge number of paths (and nodes) in the ART, which are explored independently of each other.

We propose a novel, broader view on ART-based software model checking, where a much more compact abstract space is used, resulting thus in a much smaller number of paths to be enumerated in the ART. Instead of using edges that represent single program operations, we encode entire parts of the program in one edge. In contrast to SBE, we call our new approach *large-block encoding* (LBE). In general, the new encoding may result in an exponential reduction of the number of ART nodes.

The generalization from SBE to LBE has two main consequences. First, LBE requires a more general representation of abstract states than SBE. SBE is typically based on mere *conjunctions* of predicates. Because the LBE approach summarizes large portions of the control flow, conjunctions are not sufficient, and we need to use *arbitrary Boolean combinations* of predicates to represent the abstract states. Second, LBE requires a more accurate abstraction in the abstract-successor computations. Intuitively, an abstract edge represents many different paths of the program, and therefore it is necessary that the abstract-successor computations take the relationships between the predicates into account.

In order to make this generalization practical, we rely on efficient solvers for satisfiability modulo theories (SMT). In particular, enabling factors are the capability of performing Boolean reasoning efficiently (e.g., [18]), the availability of effective algorithms for abstraction computation (e.g., [8], [15]), and interpolation procedures to extract new predicates [6], [9].

Considering Boolean abstraction and large-block encoding in addition to the traditional techniques, we obtain the following interesting observations: (i) whilst the SBE approach requires a large number of successor computations, the LBE approach reduces the number of successor computations dramatically (possibly exponentially); (ii) whilst Cartesian abstraction can be efficiently computed with a linear number of SMT solver queries, Boolean abstraction is expensive to compute because it requires an enumeration of all satisfiable assignments for the predicates. Therefore, two combinations of the above strategies provide an interesting tradeoff: The combination of SBE with Cartesian abstraction was successfully implemented by tools like BLAST and SLAM. We investigate the combination of LBE with Boolean abstraction, by first formally defining LBE in terms of a summarization of the control-flow automaton for the program, and then implementing this LBE approach together with a Boolean predicate abstraction. We evaluate the performance and precision by comparing it with the model checker BLAST and with an own implementation of the traditional approach. Our own implementation of

[1] Technical Report SFU-CS-2009-09, DISI-09-026, FBK-irst-2009.04.005.

[2] Supported in part by the Canadian NSERC grant RGPIN 341819-07 and by the SFU grant PRG 06-3. [3] Supported in part by the European Commission grant FP7-2007-IST-1-217069 COCONUT. [4] Supported in part by the SRC/GRC grant 2009-TJ-1880 WOLFLING and by the MIUR grant PRIN 20079E5KM8_002.

the SBE and LBE approach is integrated as a new component into CPACHECKER [5][1]. The experiments show that our new approach outperforms the previous approach.

Example. We illustrate the advantage of LBE over SBE on the example program in Fig. 1 (a). In SBE, each program location is modeled explicitly, and an abstract-successor computation is performed for each program operation. Figure 1 (b) shows the structure of the resulting ART. In the figure, abstract states are drawn as ellipses, and labeled with the location of the abstract state; the arrows indicate that there exists an edge from the source location to the target location in the control-flow. The ART represents all feasible program paths. For example, the leftmost program path is taking the 'then' branch of every 'if' statement. For every edge in the ART, an abstract-successor computation is performed, which potentially includes several SMT solver queries. The problems given to the SMT solver are usually very small, and the runtime sums up over a large amount of simple queries. Therefore, model checkers that are based on SBE (like BLAST) experience serious performance problems on programs with such an exploding structure (cf. the `test_locks` examples in Table I). In LBE, the control-flow graph is summarized, such that control-flow edges represent entire subgraphs of the original control-flow. In our example, most of the program is summarized into one control-flow edge. Figure 1 (c) shows the structure of the resulting ART, in which all the feasible paths of the program are represented by a single edge. The exponential growth of the ART does not occur. □

Related Work. The model checkers SLAM and BLAST are typical examples for the SBE approach [3], [4], both based on counterexample-guided abstraction refinement (CEGAR) [10]. Also the tool SATABS is based on CEGAR, but it performs a fully symbolic search in the abstract space [12]. In contrast, our approach still follows the lazy-abstraction paradigm [14], but it abstracts and refines chunks of the program "on-the-fly". The work of McMillan is also based on lazy abstraction, but instead of using predicate abstraction for the abstract domain, Craig interpolants from infeasible error paths are directly used, thus avoiding abstract-successor computations [16]. A fundamentally different approach to software model checking is bounded model checking (BMC), with the most prominent example CBMC [11]. Programs are unrolled up to a given depth, and a formula is constructed which is satisfiable iff one of the considered program executions reaches a certain error location. The analysis tool CALYSTO is an example of an "extended static checker", following an approach similar to BMC when generating verification conditions [1], while possibly abstracting away some irrelevant parts of the program. The BMC approaches are targeted towards discovering bugs, and can not be used to prove program safety.

Structure. Section II provides the necessary background. Section III explains our contribution in detail. We experimentally evaluate our novel approach in Sect. IV. In Sect. V, we draw some conclusions and outline directions for future research.

II. Background

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.[2] 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 variables that occur in operations from Ops is denoted by X. A *program* $P = (A, l_0, l_E)$ consists of a CFA $A = (L, G)$ (which models the control flow of the program), an initial program location $l_0 \in L$ (which models the program entry) such that G does not contain any edge $(\cdot, \cdot, l_0)$, and a target program location $l_E \in L$ (which models the error location).

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 S of all data states c that imply it (i.e. $S = \{c \mid c \models \varphi\}$). 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 all concrete states: $\{(l, c) \mid c \models \varphi\}$. The *concrete semantics* of an operation $op \in Ops$ is defined by the strongest postcondition operator SP_{op}: for a formula φ, $\mathsf{SP}_{op}(\varphi)$ represents the set of data states that are reachable from any of the states in 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 $\mathsf{SP}_{s:=e}(\varphi) = \exists \hat{s} : \varphi_{[s \mapsto \hat{s}]} \wedge (s = e_{[s \mapsto \hat{s}]})$; and for an assume operation $assume(p)$, we have $\mathsf{SP}_{assume(p)}(\varphi) = \varphi \wedge p$.

A *path* σ is a sequence $\langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ of pairs of operations and locations. The path σ is called *program path* if for every i with $1 \leq i \leq n$ 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 path* $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ is defined as the successive application of the strongest post-operator for each operation: $\mathsf{SP}_{\sigma}(\varphi) = \mathsf{SP}_{op_n}(\ldots\mathsf{SP}_{op_1}(\varphi)\ldots)$. 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.

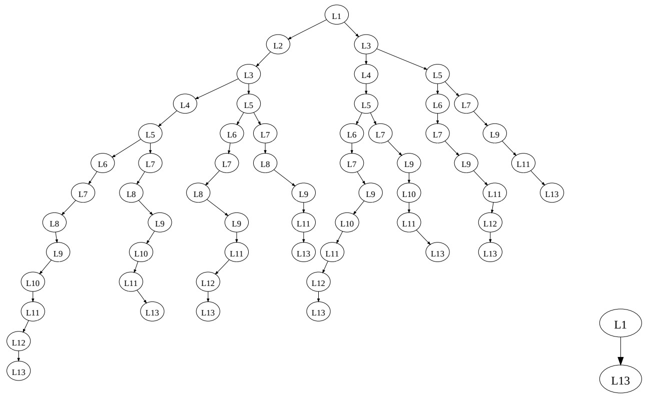
B. Predicate Abstraction

Let $\mathscr{P}$ be a set of predicates over program variables in a quantifier-free theory $\mathscr{T}$. A *formula* φ is a Boolean combi-

[1] Available at `http://www.cs.sfu.ca/~dbeyer/CPAchecker`

[2] Our implementation is based on CPACHECKER, which operates on C programs that are given in the CIL intermediate language [17]; function calls are supported.

```
L1:  if(p1) {
L2:    x1 = 1;
     }
L3:  if(p2) {
L4:    x2 = 2;
     }
L5:  if(p3) {
L6:    x3 = 3;
     }
L7:  if(p1) {
L8:    if (x1 != 1) goto ERR;
     }
L9:  if (p2) {
L10:   if (x2 != 2) goto ERR;
     }
L11: if (p3) {
L12:   if (x3 != 3) goto ERR;
     }
L13: return EXIT_SUCCESS;
ERR: return EXIT_FAILURE;
```



(a) Example C program (b) ART for SBE (c) ART for LBE

Fig. 1. Example program and corresponding ARTs for SBE and LBE; this example was mentioned as verification challenge for ART-based approaches by several colleagues.

nation of predicates from $\mathscr{P}$. A *precision for a formula* is a finite subset $\pi \subset \mathscr{P}$ of predicates.

Cartesian Predicate Abstraction. Let π be a precision. The *Cartesian predicate abstraction* $\varphi_{\mathbb{C}}^{\pi}$ *of a formula* φ is the strongest conjunction of predicates from π entailed by φ: $\varphi_{\mathbb{C}}^{\pi} := \bigwedge \{p \in \pi \mid \varphi \Rightarrow p\}$. Such a predicate abstraction of a formula φ that represents a region of concrete program states, is used as an *abstract state* (i.e., an abstract representation of the region) in program verification. For a formula φ and a precision π, the Cartesian predicate abstraction $\varphi_{\mathbb{C}}^{\pi}$ of φ can be computed by $|\pi|$ SMT-solver queries. The abstract strongest postoperator SP^{π} for a predicate abstraction π transforms the abstract state $\varphi_{\mathbb{C}}^{\pi}$ into its successor $\varphi'^{\pi}_{\mathbb{C}}$ for a program operation op, written as $\varphi'^{\pi}_{\mathbb{C}} = \mathsf{SP}^{\pi}_{op}(\varphi^{\pi}_{\mathbb{C}})$, if $\varphi'^{\pi}_{\mathbb{C}}$ is the Cartesian predicate abstraction of $\mathsf{SP}_{op}(\varphi^{\pi}_{\mathbb{C}})$, i.e., $\varphi'^{\pi}_{\mathbb{C}} = (\mathsf{SP}_{op}(\varphi^{\pi}_{\mathbb{C}}))^{\pi}_{\mathbb{C}}$. For more details, we refer the reader to the work of Ball et al. [2].

Boolean Predicate Abstraction. Let π be a precision. The *Boolean predicate abstraction* $\varphi_{\mathbb{B}}^{\pi}$ *of a formula* φ is the strongest Boolean combination of predicates from π that is entailed by φ. For a formula φ and a precision π, the Boolean predicate abstraction $\varphi_{\mathbb{B}}^{\pi}$ 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 an SMT 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. The abstract strongest postoperator SP^{π} for a predicate abstraction π transforms the abstract state $\varphi_{\mathbb{B}}^{\pi}$ into its successor $\varphi'^{\pi}_{\mathbb{B}}$ for a program operation op, written as $\varphi'^{\pi}_{\mathbb{B}} = \mathsf{SP}^{\pi}_{op}(\varphi^{\pi}_{\mathbb{B}})$, if $\varphi'^{\pi}_{\mathbb{B}}$ is the Boolean predicate abstraction of $\mathsf{SP}_{op}(\varphi^{\pi}_{\mathbb{B}})$, i.e., $\varphi'^{\pi}_{\mathbb{B}} = (\mathsf{SP}_{op}(\varphi^{\pi}_{\mathbb{B}}))^{\pi}_{\mathbb{B}}$. For more details, we refer the reader to the work of Lahiri et al. [15].

C. ART-based Software Model Checking with SBE

The *precision for a program* is a function $\Pi : L \rightarrow 2^{\mathscr{P}}$, which assigns to each program location a precision for a formula. An ART-based algorithm for software model checking takes an initial precision Π (which is typically very coarse) for the predicate abstraction, and constructs an ART for the input program and Π. An ART is a tree whose nodes are labeled with program locations and abstract states [4] (i.e., $n = (l, \varphi)$). For a given ART node, all children nodes are labeled with successor locations and abstract successor states, according to the strongest postoperator and the predicate abstraction. A node $n = (l, \varphi)$ is called *covered* if there exists another ART node $n' = (l, \varphi')$ that entails n (i.e., s.t. $\varphi' \models \varphi$). An ART is called *complete* if every node is either covered or all possible abstract successor states are present in the ART as children of the node. If a complete ART is constructed and the ART does not contain any error node, then the program is considered correct [14]. If the algorithm adds an error node to the ART, then the corresponding path σ is checked to determine if σ is feasible (i.e., if the corresponding concrete program path is executable) or infeasible (i.e., if there is no corresponding program execution). In the former case the path represents a witness for a program bug. In the latter case the path is analyzed, and a refinement Π' of Π is generated, such that the same path cannot occur again during the ART exploration. The concept of using an infeasible error path for abstraction refinement is called counterexample-guided abstraction refinement (CEGAR) [10]. The concept of iteratively constructing an ART and refining only the precisions along the considered path is called lazy abstraction [14]. Craig interpolation is a successful approach to predicate extraction for refinement [13]. After the refining the precision, the algorithm continues with the next iteration, using Π' instead of Π to construct the ART, until either a complete error-free ART is obtained, or an error is found (note that the procedure might not terminate). For more details and a more in-depth illustration of the overall ART algorithm, we refer the reader to the BLAST article [4].

In order to make the algorithm scale on practical examples, implementations such as BLAST or SLAM use the simple but coarse Cartesian abstraction, instead of the expensive but precise Boolean abstraction. Despite its potential imprecision, Cartesian abstraction has been proved successful for the verification of many real-world programs. In the SBE approach, given the large number of successor computations, the computation of the Boolean predicate abstraction is in fact too expensive, as it may require an SMT solver to enumerate an exponential number of assignments on the predicates in the precision, for each single successor computation. The reason for the success of Cartesian abstraction if used together with SBE, is that for a given program path, state overapproximations that are expressible as conjunctions of atomic predicates —for which Boolean and Cartesian abstractions are equivalent— are often good enough to prove that the error location is not reachable in the abstract space.

III. Large-Block Encoding

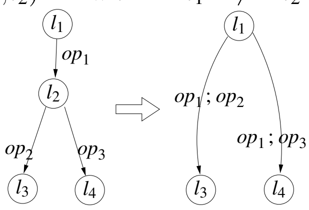
A. Summarization of Control-Flow Automata

The first, main step of LBE is the summarization of the program CFA, in which each large control-flow subgraph that is free of loops is replaced by a single control-flow edge with a large formula that represents the removed subgraph. This process, which we call *summarization* of the CPA, consists of the fixpoint application of three rewriting rules that we describe below: first we apply Rule 0 once, and then we repeatedly apply Rules 1 and 2, until no rule is applicable anymore.

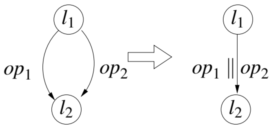
Let $P = (A, l_0, l_E)$ be a program with CFA $A = (L, G)$.

Rule 0 (Error Sink). We remove all edges $(l_E, \cdot, \cdot)$ from G, i.e., the target location l_E becomes a sink node with no outgoing edges.

Rule 1 (Sequence). If G contains an edge (l_1, op_1, l_2) with $l_1 \neq l_2$ and no other incoming edges for l_2 (i.e. edges $(\cdot, \cdot, l_2)$), and $G^{\rightarrow}_{l_2}$ is the subset of G of outgoing edges for l_2, then we change the CFA A in the following way: (1) we remove location l_2 from L, and (2) we remove the edges (l_1, op_1, l_2) and all the edges in $G^{\rightarrow}_{l_2}$ from G, and for each edge $(l_2, op_i, l_i) \in G^{\rightarrow}_{l_2}$, we add the edge $(l_1, op_1\,;op_i, l_i)$ to G, where $\mathsf{SP}_{op_1;op_i}(\varphi) = \mathsf{SP}_{op_i}(\mathsf{SP}_{op_1}(\varphi))$. (Note that $G^{\rightarrow}_{l_2}$ might contain an edge $(l_2, \cdot, l_1)$.)



Rule 2 (Choice). If $L_2 = \{l_1, l_2\}$ and $A_{|L_2} = (L_2, G_2)$ is the subgraph of A with nodes from L_2 and the set G_2 of edges contains the two edges (l_1, op_1, l_2) and (l_1, op_2, l_2), then we change the CFA A in the following way: (1) we remove the two edges (l_1, op_1, l_2) and (l_1, op_2, l_2) from G and add the edge $(l_1, op_1 \parallel op_2, l_2)$ to G, where $\mathsf{SP}_{op_1 \| op_2}(\varphi) = \mathsf{SP}_{op_1}(\varphi) \vee \mathsf{SP}_{op_2}(\varphi)$. (Note that there might be a backwards edge $(l_2, \cdot, l_1)$.)



Let $P = (A, l_0, l_E)$ be a program and let A' be another CFA for P. The CFA A' is the *summarization* of A if A' is obtained from A via stepwise application of the two rules, and none of the two rules can be further applied.

Example. Figure 2 shows a program (a) and its corresponding CFA (b). The control-flow automaton (CFA) is stepwise transformed to its summarization CFA (h) as follows: Rule 1 eliminates location 6 to (c), Rule 1 eliminates location 3 to (d), Rule 1 eliminates location 4 to (e), Rule 2 eliminates one edge 2–5 to (f), Rule 1 eliminates location 5 to (g), Rule 1 eliminates location 2 to (h). □

In the context of this article, we use the summarization CFA for program analysis, i.e., we want to verify if an error state of the program is reachable. The following theorem, which is proved in Appendix A, states that our summarization of a CFA is correct in this sense.

Theorem 3.1 (Correctness of Summarization): Let $P = (A, l_0, l_E)$ be a program and let $A' = (L', G')$ be the summarization of A. Then: (i) $\{l_0, l_E\} \subseteq L'$, and (ii) l_E is reachable in (A', l_0, l_E) if and only if l_E is reachable in P.

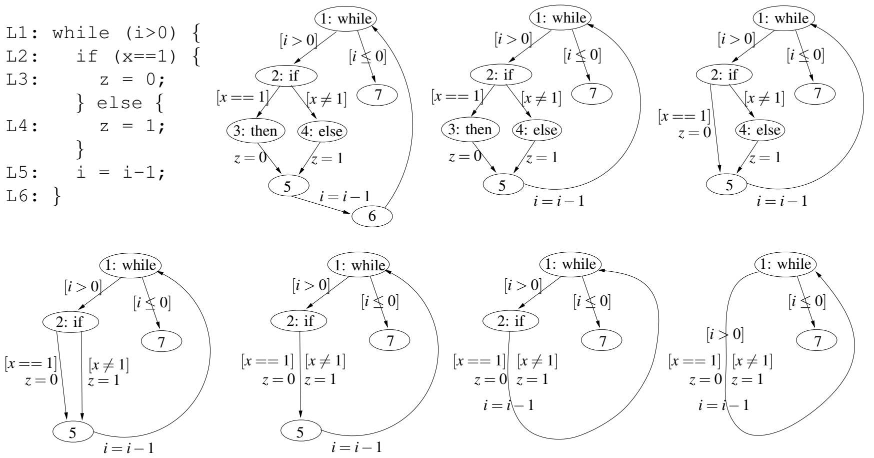
The summarization can be performed in polynomial time. The time taken by Rule 0 is proportional to the number of outgoing edges for l_E. Since each application of Rule 1 or Rule 2 removes at least one edge, there can be at most $|G| - 1$ such applications. A naive way to determine the set of locations and edges to which to apply each rule requires $O(|V| \cdot k)$ time, where k is the maximum out-degree of locations. Finally, each application of Rule 2 requires $O(1)$ time, and each application of Rule 1 $O(k)$ time. Therefore, a naive summarization algorithm requires $O(|G| \cdot |V| \cdot k)$ time, which reduces to $O(|G| \cdot |V|)$ if k is bounded (i.e., if we rewrite a priori all `switch`es into nested `if`s).[3]

B. LBE versus SBE for Software Model Checking

The use of LBE instead of the standard SBE requires no modification to the general model-checking algorithm, which is still based on the construction of an ART with CEGAR-based refinement. The main difference is that the LBE has no one-to-one correspondence between ART paths and syntactical program paths. A single CFA edge corresponds to a *set of paths* between its source and target location, and a single ART path corresponds to a *set of program paths*; an ART node represents an overapproximation of the data region that is reachable by following *any* of the program paths represented by the ART path that leads to it. This difference leads to two observations.

First, LBE can lead to exponentially-smaller ARTs than SBE, and thus it can drastically reduce the number successor computations (cf. example in Sect. I) and the number of abstraction-refinement steps for infeasible error paths. Each of these operations, however, is typically more expensive than with SBE, because the formulas involved are larger and have a more complex structure.

[3] In our implementation, we use a more efficient algorithm, which we do not describe here for lack of space.



```
L1: while (i>0) {
L2:   if (x==1) {
L3:     z = 0;
      } else {
L4:     z = 1;
      }
L5:   i = i-1;
L6: }
```

Fig. 2. CFA Transformation: a) Program, b) CFA, c)–g) Intermediate CFAs, h) Summarization CFA. In the CFAs, *assume*(p) is represented as $[p]$, $op_1\,;op_2$ is represented by putting op_2 under op_1, and $op_1 \parallel op_2$ by putting op_2 beside op_1.

Second, LBE requires a more general representation of abstract states. When using SBE, abstract states are typically represented as sets/conjunctions of predicates. This is sufficient for practical examples because each abstract state represents a data region reachable by a single program path, which can be encoded essentially as a conjunction of atomic formulas. With LBE, such representation would be too coarse, since each abstract state represents a data region that is reachable on several different program paths. Therefore, we need to use a representation for arbitrary (and larger) Boolean combinations of predicates. This generalization of the representation of the abstract state requires a generalization of the representation of the transition, i.e., the replacement of the Cartesian abstraction with a more precise form of abstraction. In this paper, we evaluate the use of the Boolean abstraction, which allows for a precise representation of arbitrary Boolean combinations of predicates.

With respect to the traditional SBE approach, LBE allows us to trade part of the cost of the *explicit* enumeration of program paths with that of the *symbolic* computation of abstract successor states: rather than having to build large ARTs by performing a substantial amount of relatively cheap operations (Cartesian abstract postoperator applications along single edges and counterexample analysis of individual program paths), with LBE we build smaller ARTs by performing more expensive symbolic operations (Boolean abstract postoperator applications along large portions of the control flow and counterexample analysis of multiple program paths), involving formulas with a complex Boolean structure. With LBE, the *cost* of each symbolic operation, rather than their *number*, becomes a critical performance factor.

To this extent, LBE makes it possible to fully exploit the power and functionality of modern SMT solvers: First, the capability of modern SMT solvers of performing large amounts of Boolean reasoning allows for handling possibly-big Boolean combinations of atomic expressions, instead of simple conjunctions. Second, the capability of some SMT solvers to perform All-SMT and interpolant computation (see, e.g., [7]) allows for effectively performing SMT-based Boolean abstraction computation [8], [15] and interpolation-based counterexample analysis [9] respectively, which was shown to outperform previous approaches, especially when dealing with complex formulas. With SBE, instead, the use of modern SMT technology does not lead to significant improvements of the whole ART-based algorithm, because each SMT query involves simple (and often small) conjunctions only.

IV. Performance Evaluation

Implementation. In order to evaluate the proposed verification method, we integrate our algorithm as a new component into the configurable software verification toolkit CPACHECKER [5]. This implementation is written in JAVA. All example programs are preprocessed and transformed into the simple intermediate language CIL [17]. For parsing C programs, CPACHECKER uses a library from the Eclipse C/C++ Development Kit. For efficient querying of formulas in the quantifier-free theory of rational linear arithmetic and equality with uninterpreted function symbols, we leverage the SMT solver MATHSAT [7], which is integrated as a library (written in C++). We use binary decision diagrams (BDDs) for the representation of abstract-state formulas.

We run all experiments on a 1.8 GHz Intel Core2 machine with 2 GB of RAM and 2 MB of cache, running GNU/Linux. We used a timeout of 1 800 s and a memory limit of 1.8 GB.

Example Programs. We use three categories of benchmark programs. First, we experiment with programs that are specifically designed to cause an exponential blowup of the ART when using SBE (`test_locks*`, in the style of the example in Sect. I). Second, we use the device-driver programs that were previously used as benchmarks in the BLAST project. Third, we solve various verification problems for the SSH client and server software (`s3_clnt*` and `s3_srvr*`), which share the same program logic, but check different safety properties. The safety property is encoded as conditional calls of a failure location and therefore reduces to the reachability of a certain error location. All benchmarks programs from the BLAST web page are preprocessed with Cil. For the second and third groups of programs, we also performed experiments with artificial defects introduced.

Experimental Configurations. For a careful and fair performance comparison, we run experiments on three different configurations. First, we use Blast, version 2.5, which is a highly optimized state-of-the-art software model checker. Blast is implemented in the programming language OCaml. We run Blast using all four combinations of breadth-first search (`-bfs`) versus depth-first search (`-dfs`), both with and without heuristics for improving the predicate discovery. Blast provides five different levels of heuristics for predicate discovery, and we use only the lowest (`-predH 0`) and the highest option (`-predH 7`). Interestingly, every combination is best for some particular example programs, with considerable differences in runtime and memory consumption. The configuration using `-dfs -predH 7` is the winner (in terms of solved problems and total runtime) for the programs without defects, but is not able to verify four example programs (timeout). In the performance table, we provide results obtained using this configuration (column `-dfs -predH 7`), and also the best result among the four configurations for every single instance (column `best result`). For the unsafe programs, `-bfs -predH 7` performs best. All four configurations use the command-line options `-craig 2 -nosimplemem -alias ""`, which specify that Blast runs with lazy, Craig-interpolation-based refinement, no Cil preprocessing for memory access, and without pointer analysis. In all experiments with Blast, we use the same interpolation procedure (MathSAT) as in our CPAchecker-based implementation. (The results of all four configurations are provided in Appendix B, to the reviewers.)

Second, in order to separate the optimization efforts in Blast from the conceptual essence of the traditional lazy abstraction algorithm, we developed a re-implementation of the traditional algorithms as described in the Blast tool article [4]. This re-implementation is integrated as component into CPAchecker, so that the difference between SBE and LBE is only in the algorithms, not in the environment (same parser, same BDD package, same query optimization, etc.). Our SBE implementation uses a DFS algorithm. This column is labeled as SBE.

Third, we run the experiments using our new LBE algorithm, which is also implemented within CPAchecker. Our LBE implementation uses a DFS algorithm. This column is labeled as LBE. Note that the purpose of our experiments is to give evidence of the performance difference between SBE and LBE, because these two settings are closest to each other, since SBE and LBE differ only in the CFA summarization and Boolean abstraction. The other two columns are provided to give evidence that the new approach beats the highly optimized traditional implementation Blast.

We actually configured and ran experiments with all four combinations: SBE versus LBE, and Cartesian versus Boolean abstraction. The experimentation clearly showed that SBE does not benefit from Boolean abstraction in terms of precision, with substantial degrade in performance: the only programs for which it terminated successfully were the first five instances of the `test_locks` group. Similarly, the combination of LBE with Cartesian abstraction fails to solve any of the experiments, due to loss of precision. Thus, we report only on the two successful configurations, i.e., SBE in combination with Cartesian abstraction, and LBE in combination with Boolean abstraction.

TABLE III
PERFORMANCE RESULTS, PROGRAMS WITH ARTIFICIAL BUGS.

	Blast	**CPAchecker**	
Program	`(best result)`	**SBE**	**LBE**
cdaudio.BUG.i.cil.c	18.79	74.39	**9.85**
diskperf.BUG.i.cil.c	889.79	26.53	**6.78**
floppy.BUG.i.cil.c	119.60	36.49	**4.30**
kbfiltr.BUG.i.cil.c	46.80	75.45	**11.52**
parport.BUG.i.cil.c	**1.67**	14.62	2.64
s3_clnt.blast.01.BUG.i.cil.c	8.84	1514.90	**3.33**
s3_clnt.blast.02.BUG.i.cil.c	9.02	843.42	**3.27**
s3_clnt.blast.03.BUG.i.cil.c	6.64	780.72	**2.61**
s3_clnt.blast.04.BUG.i.cil.c	9.78	724.04	**3.18**
s3_srvr.blast.01.BUG.i.cil.c	7.59	MO	**2.09**
s3_srvr.blast.02.BUG.i.cil.c	7.16	>1800.00	**2.10**
s3_srvr.blast.03.BUG.i.cil.c	7.42	>1800.00	**2.08**
s3_srvr.blast.04.BUG.i.cil.c	7.33	>1800.00	**1.93**
s3_srvr.blast.06.BUG.i.cil.c	39.81	MO	**5.08**
s3_srvr.blast.07.BUG.i.cil.c	310.84	>1800.00	**28.35**
s3_srvr.blast.08.BUG.i.cil.c	40.51	>1800.00	**36.47**
s3_srvr.blast.09.BUG.i.cil.c	265.48	>1800.00	**4.94**
s3_srvr.blast.10.BUG.i.cil.c	40.24	>1800.00	**12.01**
s3_srvr.blast.11.BUG.i.cil.c	49.05	>1800.00	**4.80**
s3_srvr.blast.12.BUG.i.cil.c	38.66	>1800.00	**6.11**
s3_srvr.blast.13.BUG.i.cil.c	251.56	>1800.00	**15.20**
s3_srvr.blast.14.BUG.i.cil.c	39.94	1656.54	**4.63**
s3_srvr.blast.15.BUG.i.cil.c	40.19	>1800.00	**10.19**
s3_srvr.blast.16.BUG.i.cil.c	39.54	>1800.00	**5.21**
TOTAL (solved/time)	**24 / 2296.25**	**10 / 5747.10**	**24 / 188.67**

Discussion of Evaluation Results. Tables I and III present performance results of our experiments, for the safe and unsafe programs respectively. All runtimes are given in seconds of processor time, '>1800.00' indicates a timeout, 'MO' indicates an out-of-memory. Table II shows statistics about the algorithms for SBE and LBE only.

The first group of experiments in Table I shows that the time complexity of SBE (and Blast) can grow exponentially in the number of nested conditional statements, as expected. Table II explains why the SBE approach exhausts the memory: the number of abstract nodes in the reachability tree grows exponentially in the number of nested conditional statements. Therefore, this approach does not scale. The LBE approach reduces the loop-free part of the branching control-flow structure to a few edges (cf. example in the introduction), and the size of the ART is constant, because only the structure inside the body of the loop changes. There are no refinement steps necessary in the LBE approach, because the edges to the error location are infeasible. Therefore, no predicates are used. The runtime of the LBE approach slightly increases with the size of the program, because the formulas that are sent to the SMT solver are slightly increasing. Although in principle the complexity of the SMT problem grows exponentially in the size of the formulas, the heuristics used by SMT solvers

TABLE I
PERFORMANCE RESULTS

	BLAST		CPACHECKER	
Program	`(best result)`	`(-dfs -predH 7)`	**SBE**	**LBE**
test_locks_5.c	4.50	4.96	4.01	**0.29**
test_locks_6.c	7.81	8.81	7.22	**0.32**
test_locks_7.c	13.91	15.15	12.63	**0.34**
test_locks_8.c	25.00	26.49	23.93	**0.57**
test_locks_9.c	46.84	49.29	52.04	**0.38**
test_locks_10.c	94.57	97.85	131.39	**0.40**
test_locks_11.c	204.55	208.78	MO	**0.70**
test_locks_12.c	529.16	533.97	MO	**0.46**
test_locks_13.c	1229.27	1232.87	MO	**0.49**
test_locks_14.c	>1800.00	>1800.00	MO	**0.50**
test_locks_15.c	>1800.00	>1800.00	MO	**0.56**
cdaudio.i.cil.c	175.76	264.12	MO	**53.55**
diskperf.i.cil.c	>1800.00	>1800.00	MO	**232.00**
floppy.i.cil.c	218.26	>1800.00	MO	**56.36**
kbfiltr.i.cil.c	23.55	32.80	41.12	**7.82**
parport.i.cil.c	738.82	915.79	MO	**378.04**
s3_clnt.blast.01.i.cil.c	33.01	1000.41	755.81	**19.51**
s3_clnt.blast.02.i.cil.c	62.65	312.77	1075.45	**16.00**
s3_clnt.blast.03.i.cil.c	60.62	314.74	746.31	**49.50**
s3_clnt.blast.04.i.cil.c	63.96	197.65	730.80	**25.45**
s3_srvr.blast.01.i.cil.c	811.27	1036.89	>1800.00	**125.33**
s3_srvr.blast.02.i.cil.c	360.47	360.47	>1800.00	**122.83**
s3_srvr.blast.03.i.cil.c	276.19	276.19	>1800.00	**98.47**
s3_srvr.blast.04.i.cil.c	175.64	301.85	>1800.00	**71.77**
s3_srvr.blast.06.i.cil.c	304.63	304.63	>1800.00	**59.70**
s3_srvr.blast.07.i.cil.c	478.05	666.53	>1800.00	**85.82**
s3_srvr.blast.08.i.cil.c	115.76	115.76	>1800.00	**61.29**
s3_srvr.blast.09.i.cil.c	445.21	1037.09	>1800.00	**126.47**
s3_srvr.blast.10.i.cil.c	115.10	115.10	>1800.00	**63.36**
s3_srvr.blast.11.i.cil.c	367.98	844.28	>1800.00	**162.76**
s3_srvr.blast.12.i.cil.c	304.05	304.05	>1800.00	**170.33**
s3_srvr.blast.13.i.cil.c	580.33	878.54	>1800.00	**74.49**
s3_srvr.blast.14.i.cil.c	303.21	303.21	>1800.00	**50.38**
s3_srvr.blast.15.i.cil.c	115.88	115.88	>1800.00	**21.01**
s3_srvr.blast.16.i.cil.c	305.11	305.11	>1800.00	**127.82**
TOTAL (solved/time)	**32 / 8591.12**	**31 / 12182.03**	**11 / 3580.71**	**35 / 2265.07**
TOTAL w/o `test_locks*`	**23 / 6435.51**	**22 / 10003.06**	**5 / 3349.48**	**24 / 2260.07**

avoid the exponential enumeration that we observe in the case of SBE.

For the two other classes of experiments, we see that LBE is able to successfully complete all benchmarks, and shows significant performance gains over SBE. SBE is able to solve only about one third of all benchmarks, and for the ones that complete, it is clearly outperformed by LBE. In Table II, we see that SBE has in general a much larger ART. In Table I we observe not only that LBE performs significantly better than the `-dfs -predH 7` configuration of BLAST, but that LBE is better than any BLAST configuration (column `best result`). LBE performed best also in finding the error paths (cf. Table III), clearly outperforming both SBE and BLAST.

In summary, the experiments show that the LBE approach outperforms the SBE approach, both for correct and defective programs. This provides evidence of the benefits of a "more symbolic" analysis as performed in the LBE approach. One might argue that our CPACHECKER-based SBE implementation might be sub-optimal although it uses the same implementation and execution environment as LBE; this is why we compare with BLAST as well, and the experiments become even more impressive when considering that BLAST is the result of several years of fine-tuning.

V. Conclusion and Future Work

We have proposed LBE as an alternative to the SBE model-checking approach, based on the idea that transitions in the abstract space represent larger fragments of the program. Our novel approach results in significantly smaller ARTs, where abstract successor computations are more involved, and thus trading cost of many explicit enumerations of program paths with the cost of symbolic successor computations. A thorough experimental evaluation shows the advantages of LBE against both our implementation of SBE and the state-of-the-art BLAST system.

In our future work, we plan to implement McMillan's interpolation-based lazy-abstraction approach [16], and experiment with SBE versus LBE versions of his algorithm. Furthermore, we plan to investigate the use of adjustable precision-based techniques for the construction of the large blocks on-the-fly (instead of the current preprocessing step).

TABLE II
DETAILED COMPARISON BETWEEN SBE AND LBE ENCODINGS; ENTRIES MARKED WITH (*) DENOTE PARTIAL STATISTICS FOR ANALYSES THAT TERMINATED UNSUCCESSFULLY (IF AVAILABLE).

	SBE					LBE				
	ART	# ref	# predicates			ART	# ref	# predicates		
Program	size	steps	Tot	Avg	Max	size	steps	Tot	Avg	Max
test_locks_5.c	1344	50	10	3	10	4	0	0	0	0
test_locks_6.c	2301	72	12	4	12	4	0	0	0	0
test_locks_7.c	3845	98	14	5	14	4	0	0	0	0
test_locks_8.c	6426	128	16	6	16	4	0	0	0	0
test_locks_9.c	10926	162	18	7	18	4	0	0	0	0
test_locks_10.c	19091	200	20	8	20	4	0	0	0	0
test_locks_11.c	24779(*)	242(*)	22(*)	9(*)	22(*)	4	0	0	0	0
test_locks_12.c	28119(*)	288(*)	24(*)	10(*)	24(*)	4	0	0	0	0
test_locks_13.c	31739(*)	338(*)	26(*)	10(*)	26(*)	4	0	0	0	0
test_locks_14.c	35178(*)	392(*)	28(*)	11(*)	28(*)	4	0	0	0	0
test_locks_15.c	38777(*)	450(*)	30(*)	12(*)	30(*)	4	0	0	0	0
cdaudio.i.cil.c	53323(*)	445(*)	147(*)	9(*)	78(*)	6909	140	79	5	16
diskperf.i.cil.c	–	–	–	–	–	4890	145	56	6	21
floppy.i.cil.c	31079(*)	301(*)	79(*)	7(*)	35(*)	9668	176	58	4	13
kbfiltr.i.cil.c	19640	153	53	5	27	1577	47	18	2	6
parport.i.cil.c	26188(*)	360(*)	143(*)	4(*)	41(*)	38488	474	168	4	17
s3_clnt.blast.01.i.cil.c	122678	557	59	20	59	36	5	47	11	47
s3_clnt.blast.02.i.cil.c	354132	532	55	19	55	36	5	51	12	51
s3_clnt.blast.03.i.cil.c	196599	534	55	19	55	39	5	75	18	75
s3_clnt.blast.04.i.cil.c	172444	538	55	19	55	36	5	47	11	47
s3_srvr.blast.01.i.cil.c	232195(*)	774(*)	70(*)	20(*)	70(*)	101	6	88	22	88
s3_srvr.blast.02.i.cil.c	254667(*)	745(*)	79(*)	19(*)	78(*)	109	7	75	18	75
s3_srvr.blast.03.i.cil.c	–	–	–	–	–	91	6	85	21	85
s3_srvr.blast.04.i.cil.c	–	–	–	–	–	103	7	82	20	82
s3_srvr.blast.06.i.cil.c	295698(*)	576(*)	63(*)	14(*)	63(*)	94	6	84	21	84
s3_srvr.blast.07.i.cil.c	–	–	–	–	–	92	5	85	21	85
s3_srvr.blast.08.i.cil.c	279991(*)	549(*)	57(*)	15(*)	57(*)	89	5	88	22	88
s3_srvr.blast.09.i.cil.c	189541(*)	720(*)	72(*)	16(*)	71(*)	193	4	72	18	72
s3_srvr.blast.10.i.cil.c	307671(*)	597(*)	55(*)	16(*)	55(*)	91	5	79	19	79
s3_srvr.blast.11.i.cil.c	–	–	–	–	–	48	6	69	17	69
s3_srvr.blast.12.i.cil.c	258546(*)	563(*)	57(*)	15(*)	57(*)	99	6	94	23	94
s3_srvr.blast.13.i.cil.c	167333(*)	682(*)	70(*)	18(*)	69(*)	90	5	81	20	81
s3_srvr.blast.14.i.cil.c	318982(*)	643(*)	65(*)	13(*)	64(*)	92	6	83	20	83
s3_srvr.blast.15.i.cil.c	279319(*)	579(*)	58(*)	15(*)	58(*)	71	4	71	17	71
s3_srvr.blast.16.i.cil.c	346185(*)	596(*)	59(*)	12(*)	58(*)	98	6	86	21	86

This would enable a dynamic adjustment of the size of the large blocks, and thus we could fine-tune the amount of work that is delegated to the SMT solver. Also, we plan to explore other techniques for computing abstract successors which are more precise than Cartesian abstraction but less expensive than Boolean abstraction.

Acknowledgments. We thank Roman Manevich for interesting discussions about BLAST's performance bottlenecks.

References

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

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

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

[4] 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*, 9(5-6):505–525, 2007.

[5] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. Technical Report SFU-CS-2009-02, Simon Fraser University, January 2009.

[6] D. Beyer, D. Zufferey, and R. Majumdar. CSISAT: Interpolation for LA+EUF. In *Proc. CAV*, LNCS 5123, pages 304–308. Springer, 2008.

[7] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MATHSAT 4 SMT solver. In *Proc. CAV*, LNCS 5123. Springer, 2008.

[8] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing predicate abstractions by integrating BDDs and SMT solvers. In *Proc. FMCAD*, pages 69–76. IEEE, 2007.

[9] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *Proc. TACAS*, LNCS 4963, pages 397–412. Springer, 2008.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[11] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, LNCS 2988, pages 168–176. Springer, 2004.

[12] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS*, LNCS 3440, pages 570–574. Springer, 2005.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

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

[16] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.

[17] 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, pages 213–228. Springer, 2002.

[18] R. Sebastiani. Lazy satisfiability modulo theories. *J. Satisfiability, Boolean Modeling and Computation*, 3, 2007.

Appendix

A. Proof of Theorem 3.1

In order to prove Theorem 3.1, we introduce some auxiliary lemmas.

Lemma A.1: Let (l, op, l') be a CFA edge, and $\{\varphi_i\}_i$ a collection of formulas. Then

$$\mathsf{SP}_{op}(\bigvee_i \varphi_i) \equiv \bigvee_i \mathsf{SP}_{op}(\varphi_i).$$

Proof: If op is an assignment operation $s := e$, then

$$\begin{aligned}
\mathsf{SP}_{s:=e}(\bigvee_i \varphi_i) &= \exists \widehat{s}.((\bigvee_i \varphi_i)_{[s \mapsto \widehat{s}]} \wedge (s = e_{[s \mapsto \widehat{s}]})) \\
&\equiv \exists \widehat{s}.(\bigvee_i (\varphi_{i[s \mapsto \widehat{s}]} \wedge (s = e_{[s \mapsto \widehat{s}]}))) \\
&\equiv \bigvee_i (\exists \widehat{s}.(\varphi_{i[s \mapsto \widehat{s}]} \wedge (s = e_{[s \mapsto \widehat{s}]}))) \\
&\equiv \bigvee_i \mathsf{SP}_{s:=e}(\varphi_i)
\end{aligned}$$

If op is an assume operation $assume(p)$, then

$$\begin{aligned}
\mathsf{SP}_{assume(p)}(\bigvee_i \varphi_i) &= (\bigvee_i \varphi_i) \wedge p \\
&\equiv \bigvee_i (\varphi_i \wedge p) \\
&\equiv \bigvee_i \mathsf{SP}_{assume(p)}(\varphi_i)
\end{aligned}$$

The remaining two cases can be proven by induction.
If $op = op_1\,; op_2$, then

$$\begin{aligned}
\mathsf{SP}_{op_1;op_2}(\bigvee_i \varphi_i) &= \mathsf{SP}_{op_2}(\mathsf{SP}_{op_1}(\bigvee_i \varphi_i)) \\
&\equiv \mathsf{SP}_{op_2}(\bigvee_i \mathsf{SP}_{op_1}(\varphi_i)) \\
&\equiv \bigvee_i \mathsf{SP}_{op_2}(\mathsf{SP}_{op_1}(\varphi_i)) \\
&\equiv \bigvee_i \mathsf{SP}_{op_1;op_2}(\varphi_i)
\end{aligned}$$

If $op = op_1 \parallel op_2$, then

$$\begin{aligned}
\mathsf{SP}_{op_1 \parallel op_2}(\bigvee_i \varphi_i) &= \mathsf{SP}_{op_1}(\bigvee_i \varphi_i) \vee \mathsf{SP}_{op_2}(\bigvee_i \varphi_i) \\
&\equiv (\bigvee_i \mathsf{SP}_{op_1}(\varphi_i)) \vee (\bigvee_i \mathsf{SP}_{op_2}(\varphi_i)) \\
&\equiv \bigvee_i (\mathsf{SP}_{op_1}(\varphi_i) \vee \mathsf{SP}_{op_2}(\varphi_i)) \\
&\equiv \bigvee_i \mathsf{SP}_{op_1 \parallel op_2}(\varphi_i)
\end{aligned}$$

□ ■

Lemma A.2: Let $A = (L, G)$ be a CFA, and let $A' = (L', G')$ be a summarization of A. Let σ be a path in A such that its initial and final locations occur also in L'. Then for all φ, there exists a path σ' in A', with the same initial and final locations as σ, such that $\mathsf{SP}_\sigma(\varphi) \models \mathsf{SP}_{\sigma'}(\varphi)$.

Proof: CFA A' is obtained from A by a sequence of n rule applications. If $n = 0$ we have $A' = A$. If the lemma holds for one rule application, we can show by induction that the lemma holds for any finite sequence of rule applications.

We now show that the lemma holds for one rule application. Let $\sigma = \sigma_1, (l_i, op_i, l_j)$. The proof is by induction on the length of σ. (The base case is when σ_1 is empty.)

If $l_i \in L'$, by the inductive hypothesis there exists a path σ_1' in A' such that $\mathsf{SP}_{\sigma_1}(\varphi) \models \mathsf{SP}_{\sigma_1'}(\varphi)$. If $(l_i, op_i, l_j) \in G'$, then we can take $\sigma' = \sigma_1', (l_i, op_i, l_j)$. Otherwise, (l_i, op_i, l_j) must have been removed by an application of Rule 2, [4] and so G' contains an edge $(l_i, op_i \parallel \cdot, l_j)$. Therefore, we can take $\sigma' = \sigma_1', (l_i, op_i \parallel \cdot, l_j)$.

If $l_i \notin L'$, then by hypothesis $\sigma \equiv \sigma_2, (l_k, op_k, l_i), (l_i, op_i, l_j)$. Moreover, l_i has been removed by an application of Rule 1. By the definition of Rule 1, (l_k, op_k, l_i) is the only incoming edge for l_i in G. Therefore, G' contains an edge $(l_k, op_k\,; op_i, l_j)$ and clearly $l_k \in L'$. Thus, by the inductive hypothesis there exists a path σ_2' in A' such that $\mathsf{SP}_{\sigma_2}(\varphi) \models \mathsf{SP}_{\sigma_2'}(\varphi)$, and so we can take $\sigma' = \sigma_2', (l_k, op_k\,; op_i, l_j)$. □ ■

Lemma A.3: Let $A = (L, G)$ be a CFA, and let $A' = (L', G')$ be a summarization of A. Let σ' be a path in A'. Then for all φ, there exists a set Σ of paths in A, with the same initial and final locations as σ', such that $\mathsf{SP}_{\sigma'}(\varphi) \equiv \bigvee_{\sigma \in \Sigma} \mathsf{SP}_\sigma(\varphi)$.

Proof: CFA A' is obtained from A by a sequence of n rule applications. If $n = 0$ we have $A' = A$. If the lemma holds for one rule application, we can show by induction that the lemma holds for any finite sequence of rule applications.

We now show that the lemma holds for one rule application. Let $\sigma' = \sigma_p', (l_i, op_i, l_j)$ be a path in A'. The proof is by induction on the length of σ'. (The base case is when σ_p' is empty.)

First, we observe that all locations in σ' occur also in G.

By the inductive hypothesis, there exists a set Σ_p of paths in A, with the same initial and final locations as σ_p', such that $\mathsf{SP}_{\sigma_p'}(\varphi) \equiv \bigvee_{\sigma_p \in \Sigma_p} \mathsf{SP}_{\sigma_p}(\varphi)$.

If $(l_i, op_i, l_j) \in G$, then we can take $\Sigma = \{\sigma_p, (l_i, op_i, l_j) \mid \sigma_p \in \Sigma_p\}$ (by Lemma A.1).

Otherwise, (l_i, op_i, l_j) was generated by an application of one of the Rules. If it was generated by Rule 1, then G contains two edges (l_i, op_i', l_k) and (l_k, op_k, l_j) such that $op_i = op_i'\,; op_k$. Then we can take $\Sigma = \{\sigma_p, (l_i, op_i', l_k), (l_k, op_k, l_j) \mid \sigma_p \in \Sigma_p\}$ (by Lemma A.1). If (l_i, op_i, l_j) was generated by Rule 2, then G contains two edges (l_i, op_i', l_j) and (l_i, op_i'', l_j) such that $op_i = op_i' \parallel op_i''$. Let $\Sigma_1 = \{\sigma_p, (l_i, op_i', l_j) \mid \sigma_p \in \Sigma_p\}$ and $\Sigma_2 = \{\sigma_p, (l_i, op_i'', l_j) \mid \sigma_p \in \Sigma_p\}$. Then we can take $\Sigma = \Sigma_1 \cup \Sigma_2$ (by Lemma A.1).

□ ■

Proof: Now we prove Theorem 3.1.

(i) The only Rule that removes locations is Rule 1. Since l_0 has no incoming edges (by definition) and l_E has no outgoing edges (because of Rule 0), they cannot be removed by Rule 1.

(ii) "→" Follows from Lemma A.2 and (i).
"←" Follows from Lemma A.3 and (i).

□ ■

B. Comparison among different BLAST configurations

[4] It could not have been removed by Rule 1, because when Rule 1 removes the edges $(\cdot, \cdot, l)$ and $(l, \cdot, \cdot)$, it removes also the location l.

TABLE IV
Comparison among different configurations of Blast. (NP indicates 'no new predicates found during refinement'.)

	Blast 1	**Blast 2**	**Blast 3**	**Blast 4**	**Blast B**
Program	`(-bfs -predH 0)`	`(-bfs -predH 7)`	`(-dfs -predH 0)`	`(-dfs -predH 7)`	`(best result)`
test_locks_5.c	8.36	8.40	**4.50**	4.96	**4.50**
test_locks_6.c	17.63	17.29	**7.81**	8.81	**7.81**
test_locks_7.c	39.90	37.83	**13.91**	15.15	**13.91**
test_locks_8.c	86.98	86.69	**25.00**	26.49	**25.00**
test_locks_9.c	173.63	189.96	**46.84**	49.29	**46.84**
test_locks_10.c	500.30	483.07	**94.57**	97.85	**94.57**
test_locks_11.c	1645.90	1534.20	**204.55**	208.78	**204.55**
test_locks_12.c	>1800.00	>1800.00	**529.16**	533.97	**529.16**
test_locks_13.c	>1800.00	>1800.00	**1229.27**	1232.87	**1229.27**
test_locks_14.c	>1800.00	>1800.00	>1800.00	>1800.00	>1800.00
test_locks_15.c	>1800.00	>1800.00	>1800.00	>1800.00	>1800.00
cdaudio.i.cil.c	380.83	475.67	**175.76**	264.12	**175.76**
diskperf.i.cil.c	–	>1800.00	NP	>1800.00	>1800.00
floppy.i.cil.c	**218.26**	>1800.00	NP	>1800.00	**218.26**
kbfiltr.i.cil.c	**23.55**	69.07	NP	32.80	**23.55**
parport.i.cil.c	**738.82**	>1800.00	NP	915.79	**738.82**
s3_clnt.blast.01.i.cil.c	72.55	526.77	**33.01**	1000.41	**33.01**
s3_clnt.blast.02.i.cil.c	80.57	268.67	**62.65**	312.77	**62.65**
s3_clnt.blast.03.i.cil.c	124.99	440.25	**60.62**	314.74	**60.62**
s3_clnt.blast.04.i.cil.c	140.60	138.75	**63.96**	197.65	**63.96**
s3_srvr.blast.01.i.cil.c	1030.27	MO	**811.27**	1036.89	**811.27**
s3_srvr.blast.02.i.cil.c	>1800.00	811.77	1088.43	**360.47**	**360.47**
s3_srvr.blast.03.i.cil.c	1166.38	424.53	961.72	**276.19**	**276.19**
s3_srvr.blast.04.i.cil.c	208.89	**175.64**	1393.08	301.85	**175.64**
s3_srvr.blast.06.i.cil.c	>1800.00	>1800.00	653.62	**304.63**	**304.63**
s3_srvr.blast.07.i.cil.c	>1800.00	>1800.00	**478.05**	666.53	**478.05**
s3_srvr.blast.08.i.cil.c	>1800.00	411.92	647.87	**115.76**	**115.76**
s3_srvr.blast.09.i.cil.c	>1800.00	1296.56	**445.21**	1037.09	**445.21**
s3_srvr.blast.10.i.cil.c	>1800.00	>1800.00	645.23	**115.10**	**115.10**
s3_srvr.blast.11.i.cil.c	1692.77	1011.15	**367.98**	844.28	**367.98**
s3_srvr.blast.12.i.cil.c	>1800.00	1188.43	658.16	**304.05**	**304.05**
s3_srvr.blast.13.i.cil.c	>1800.00	MO	**580.33**	878.54	**580.33**
s3_srvr.blast.14.i.cil.c	>1800.00	463.95	653.85	**303.21**	**303.21**
s3_srvr.blast.15.i.cil.c	>1800.00	604.01	645.35	**115.88**	**115.88**
s3_srvr.blast.16.i.cil.c	>1800.00	653.87	651.30	**305.11**	**305.11**
TOTAL (solved/time)	**19/8351.18**	**23/11318.45**	**29/13233.06**	**31/12182.03**	**32/8591.12**

TABLE V
COMPARISON AMONG DIFFERENT CONFIGURATIONS OF BLAST, PROGRAMS WITH ARTIFICIAL BUGS. (NP INDICATES 'NO NEW PREDICATES FOUND DURING REFINEMENT'.)

	BLAST 1	**BLAST 2**	**BLAST 3**	**BLAST 4**	**BLAST B**
Program	`(-bfs -predH 0)`	`(-bfs -predH 7)`	`(-dfs -predH 0)`	`(-dfs -predH 7)`	`(best result)`
cdaudio.BUG.i.cil.c	108.85	99.82	26.83	**18.79**	**18.79**
diskperf.BUG.i.cil.c	**889.79**	>1800.00	926.70	>1800.00	**889.79**
floppy.BUG.i.cil.c	**119.60**	>1800.00	127.68	>1800.00	**119.60**
kbfiltr.BUG.i.cil.c	70.83	144.25	NP	**46.80**	**46.80**
parport.BUG.i.cil.c	5.70	10.95	**1.67**	2.24	**1.67**
s3_clnt.blast.01.BUG.i.cil.c	1003.92	28.30	304.63	**8.84**	**8.84**
s3_clnt.blast.02.BUG.i.cil.c	118.48	**9.02**	131.42	12.26	**9.02**
s3_clnt.blast.03.BUG.i.cil.c	167.73	**6.64**	133.97	12.20	**6.64**
s3_clnt.blast.04.BUG.i.cil.c	187.18	**9.78**	139.04	11.70	**9.78**
s3_srvr.blast.01.BUG.i.cil.c	103.06	**7.59**	>1800.00	162.90	**7.59**
s3_srvr.blast.02.BUG.i.cil.c	123.00	**7.16**	>1800.00	183.34	**7.16**
s3_srvr.blast.03.BUG.i.cil.c	55.21	**7.42**	1434.01	49.74	**7.42**
s3_srvr.blast.04.BUG.i.cil.c	79.16	**7.33**	>1800.00	53.22	**7.33**
s3_srvr.blast.06.BUG.i.cil.c	1623.73	56.11	558.18	**39.81**	**39.81**
s3_srvr.blast.07.BUG.i.cil.c	1582.86	**310.84**	1327.50	MO	**310.84**
s3_srvr.blast.08.BUG.i.cil.c	>1800.00	73.59	530.10	**40.51**	**40.51**
s3_srvr.blast.09.BUG.i.cil.c	>1800.00	**265.48**	1284.77	MO	**265.48**
s3_srvr.blast.10.BUG.i.cil.c	>1800.00	66.88	528.29	**40.24**	**40.24**
s3_srvr.blast.11.BUG.i.cil.c	722.64	**49.05**	1515.26	207.09	**49.05**
s3_srvr.blast.12.BUG.i.cil.c	620.03	**38.66**	555.60	39.28	**38.66**
s3_srvr.blast.13.BUG.i.cil.c	831.45	**251.56**	1600.65	626.93	**251.56**
s3_srvr.blast.14.BUG.i.cil.c	773.26	53.93	557.13	**39.94**	**39.94**
s3_srvr.blast.15.BUG.i.cil.c	>1800.00	77.51	530.85	**40.19**	**40.19**
s3_srvr.blast.16.BUG.i.cil.c	973.44	55.97	558.44	**39.54**	**39.54**
TOTAL (solved/time)	**20/10159.92**	**22/1637.84**	**20/12772.72**	**20/1675.56**	**24/2296.25**