



The United Nations
University

UNU-IIST

International Institute for
Software Technology

Slicing Abstraction using Path Formulas

Evren Ermis, Jochen Hoenicke,
Andreas Podelski, and Martin Schäfer

June 2011

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

Peter Haddawy, Director



The United Nations
University

UNU-IIST

International Institute for
Software Technology

P.O. Box 3058
Macao

Slicing Abstraction using Path Formulas

Evren Ermis, Jochen Hoenicke,
Andreas Podelski, and Martin Schäf

Abstract

We present a new interpolation based abstraction refinement algorithm for software model-checking. The refinement step uses interpolants computed from an error path to split the abstract states on this path. It incorporates the ideas of *slicing abstraction* and *large block encoding*. We run our model-checker on several benchmarks and show empirically that the presented algorithm terminates more often with a result than existing model checking techniques.

Evren Ermis is a researcher at the chair of software engineering at the university of Freiburg, Germany. His research interest is in the area of model checking and program verification. ermis@informatik.uni-freiburg.de.

Jochen Hoenicke is a postdoctoral researcher at the chair of software engineering at the university of Freiburg, Germany. His research interest is in the area of model checking, theorem proving, and requirement analysis. hoenicke@informatik.uni-freiburg.de.

Andreas Podelski is a professor for software engineering at the university of Freiburg, Germany. His research interest is in the area of program analysis and verification, model checking, and requirement analysis. podelski@informatik.uni-freiburg.de.

Martin Schäfer is a postdoctoral research fellow of the rCOS group at UNU-IIST. His research interest is in the area of program verification, static analysis and runtime verification. shaef@iist.unu.edu.

Contents

1	Introduction	7
2	Example	8
2.1	Large block encoding	8
2.2	Interpolant-based model checker	10
3	Preliminaries	11
4	Program Graph Reduction	13
5	Interpolant-based Model Checking	15
6	Experimental Evaluation	18
7	Related Work	22
8	Conclusion	23

1 Introduction

Abstraction refinement gives us the possibility to prove properties of a system on an abstract model without actually expanding this model to the state level. Being able to compute a proof on compact abstract models allows us to prove properties of systems of realistic size. The challenge when refining is to modify the abstract model in a way that the desired property can be shown before the model becomes prohibitively large. The use of Craig interpolants is one promising approach to steer the refinement of an abstract model [13] towards a compact representation that is yet expressive enough to prove or refute the desired property. Craig interpolants can help to refine the abstract model relatively to the desired property as they contain only information deduced by a theorem prover in refuting counterexamples for this model. In that sense, Craig interpolants support the idea of refining an abstract model locally as needed.

Slicing abstraction is one approach to efficiently refine an abstract model using Craig interpolants. Slicing abstraction is a generalization of lazy abstraction [9, 14]. It exploits locality in nodes rather than in traces by splitting individual nodes of the abstraction. This tends to result in better interpolants and thus leads, in many cases, to a faster termination of the model checking algorithm.

While for lazy abstraction it has been shown that computing interpolants from path formulas rather than from predicates can yield significant performance improvements [14], there exists no equivalent approach for slicing abstraction so far. One obstacle is, that using path formulas obtained from the refutation of single program paths might force the algorithm to split each node many times, as the derived interpolants can refer to different parts of the program. This can result in an exponential number of splits and thus reduces the chance that the algorithm terminates dramatically.

In this technical report we investigate if the improvements made in lazy abstraction by computing the interpolants from path formulas [14] rather than from predicates, can be applied for slicing abstraction as well. We present a novel model checking algorithm based on slicing abstraction that uses Craig interpolants obtained from refuting *sets of program paths*. In a first step, our algorithm collapses non-looping subprograms into single transitions using *large-block encoding* [4]. In the resulting graph, each path corresponds to a set of program paths in the original program. Thus, we can derive interpolants from the refutation of sets of paths, which helps us to reduce the number of splits needed to an extent that our algorithm can efficiently handle programs of a realistic size.

We illustrate the model checking procedure using an example in Section 2. In Section 3 we introduce the basic definitions of program graphs, program safety, and interpolants. Section 4 presents our large-block encoding implementation. In Section 5 we present our model checking algorithm. An evaluation of the algorithm and detailed comparison with existing algorithms is given in Section 6.

2 Example

We will illustrate the approach by applying it to Program \mathcal{P} (Fig. 1). Program \mathcal{P} has several branches, which can be taken non-deterministically. Each branch increments one variable whilst decrementing a second one. Therefore, the sum of x , y , and z is equal to the initial value n in each iteration of the loop. The loop iterates until x is 0. Consequently, $n = y + z$ should hold when the loop terminates with $x = 0$. This is checked by the assert statement. From this program \mathcal{P} we derive the control flow graph \mathcal{G} (Fig. 2).

```

1  procedure main() {
2      var x,y,z,n: int;

4      assume(n == x && y == 0 && z == 0);

6      while(x != 0) {
7          if (*) {
8              x := x + 1;
9              y := y - 1;
10         }
11         if (*) {
12             y := y + 1;
13             z := z - 1;
14         }
15         if (*) {
16             x := x - 1;
17             z := z + 1;
18         }
19     }
20     assert(n == y + z);
21 }

```

Figure 1: Code of program \mathcal{P} . Non-deterministically increments one variable whilst decrementing a second variable. \mathcal{P} is safe if the assertion (Line 20) holds on every execution.

2.1 Large block encoding

Large block encoding(LBE) reduces loop-free subgraphs to single edges. Hence, checking one path in the reduced CFG covers multiple paths in the original CFG. LBE iteratively (1) reduces *sequential nodes* to single edges by using conjunctions and (2) merges *multiple edges* by using disjunctions. Via the introduced disjunctions the decision of the branching is shifted to the interpolating SMT solver. The resulting CFG consists solely of nodes that represent loop heads of the code. The edges represent contiguous loop-free code segments.

Sequential nodes This intermediate step, merges sequential nodes l_i and l_{i+1} , if l_i is the only predecessor of l_{i+1} . All dotted nodes (Fig. 2) have a single predecessor and no incoming multiple edges. Their incoming edges represent sequential code segments. The compression of such edges results in the conjunctions of the loop's transition formula (Fig. 3). E.g.

$$x'' = x + 1 \wedge y'' = y - 1$$

encodes the **then**-branch of the conditional branching in line 7 (Fig. 1).

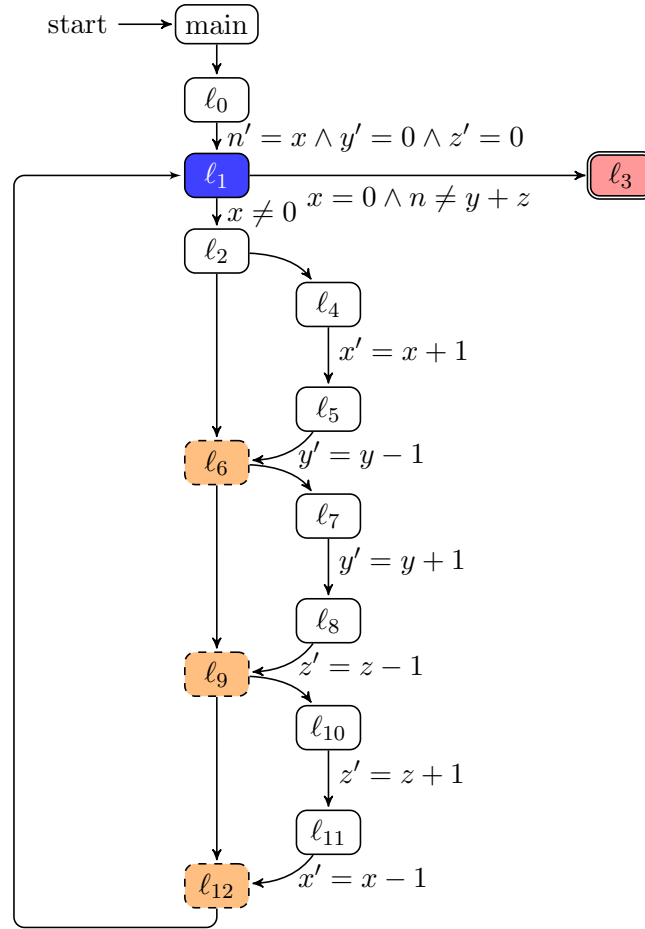


Figure 2: CFG \mathcal{G} of Figure 1. Edges without labeling carry the formula \top . l_3 represents the error state. Its guard is the negated assertion (Fig. 1, Line 20). l_1 represents the loop head (Fig. 1, Line 6). l_6 , l_9 and l_{12} are nodes that will have entering multiple edges because of the preceding branches.

Multiple edges The graph has multiple edges if the original program \mathcal{P} has conditional branchings. The branchings are reduced by joining the formulas with a disjunction. In our example (Fig. 3) the disjunctions (dashed nodes; marked orange) encode the three conditional

branchings at Line 7, 11, and 15 (Fig. 1). The left clauses of the disjunctions represent the corresponding `else`-branches. `Else`-branches must be encoded explicitly in order to preserve the variables that are changed on the alternative branch.

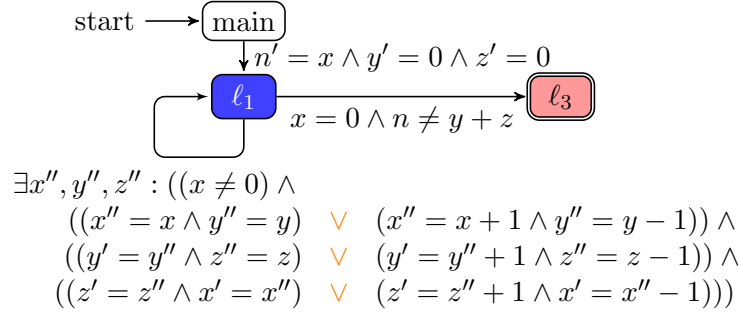


Figure 3: \mathcal{G} (Fig. 2) after reduction. The entire body of the loop is encoded in a single edge. Only the loop head (Fig. 1, Line 6) cannot be reduced any further.

ℓ_1 (solid, blue-marked node in Figure 3) is the head of the loop at Line 6. Such nodes always have at least two different predecessors. Therefore the graph cannot be compressed any further (Fig.3).

2.2 Interpolant-based model checker

Our interpolant-based software model checker uses slicing abstraction[6]. A path $\pi = (\ell_0, \dots, \ell_n)$ is encoded as a FOL formula and passed to the interpolating SMT solver. In Figure 3 the path $\pi = (main, \ell_1, \ell_3)$ has the corresponding FOL formula

$$\begin{array}{l} \text{main} \rightarrow \ell_1 \\ \ell_1 \rightarrow \ell_1 \\ \ell_1 \rightarrow \ell_3 \end{array} \quad \left(\begin{array}{l} \exists x, y''', z''', n' : ((n' = x \wedge y''' = 0 \wedge z''' = 0) \wedge \\ \left(\begin{array}{l} \exists x'', y'', z'' : ((x \neq 0) \wedge \\ ((x'' = x \wedge y'' = y''') \vee (x'' = x + 1 \wedge y'' = y''' - 1)) \wedge \\ ((y' = y'' \wedge z'' = z''') \vee (y' = y'' + 1 \wedge z'' = z''' - 1)) \wedge \\ ((z' = z'' \wedge x' = x'') \vee (z' = z'' + 1 \wedge x' = x'' - 1))) \end{array} \right) \wedge \\ (x' = 0 \wedge n' \neq y' + z') \end{array} \right)$$

The solver returns either a configuration that proves the feasibility of the path π or an array of interpolants[14]. If the path is feasible, the program is unsafe. If it is not feasible, the interpolants are used to split the path π (Fig. 4). The returned interpolant I_i (e.g. $(n' = x) \wedge (y' = 0) \wedge (z' = 0)$ in Figure 4) is appended to the corresponding node ℓ_{i+1} (ℓ_1 in Figure 4) and its negation is appended to the split node ℓ'_{i+1} . ℓ'_{i+1} inherits all incoming and outgoing edges of ℓ_{i+1} . All edges of ℓ_{i+1} and ℓ'_{i+1} are checked for feasibility. If an edge is infeasible, it is removed (dotted edges in Figure 4). In the next iteration of our interpolant-based model checker, we obtain an interpolant I_2 (e.g. $n' = x + y + z$ in Figure 5) for ℓ'_1 . In our example I_2 is an inductive invariant of the loop. The edge leading to the split node ℓ'_1 cannot be feasible. The edge from ℓ'_1 to ℓ_3 cannot be feasible either. The subgraph, consisting of ℓ'_1 and ℓ_3 , is not reachable from **start**. Hence,

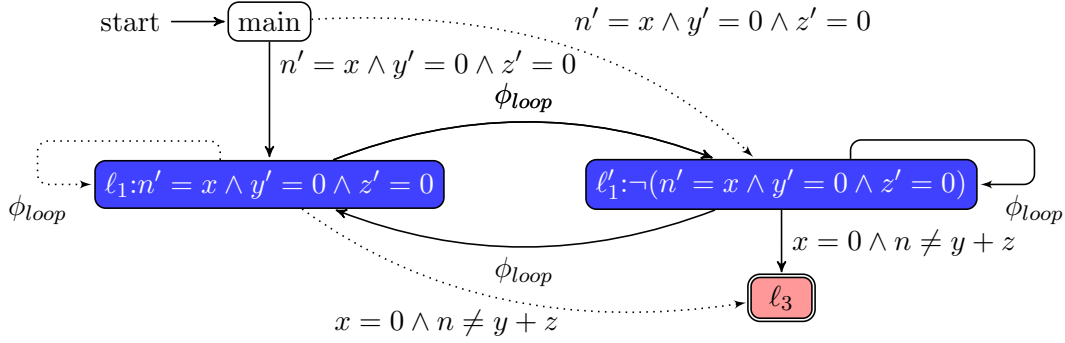


Figure 4: Graph \mathcal{G} after splitting the error path. The control flow is partitioned by appending the interpolant $n' = x \wedge y' = 0 \wedge z' = 0$ to ℓ_1 and its negation to ℓ'_1 . The label ϕ_{loop} denotes the loop formula. Dotted edges are infeasible and will be removed.

our model does not contain any error paths. The algorithm stops and has proven the safety of the program \mathcal{P} .

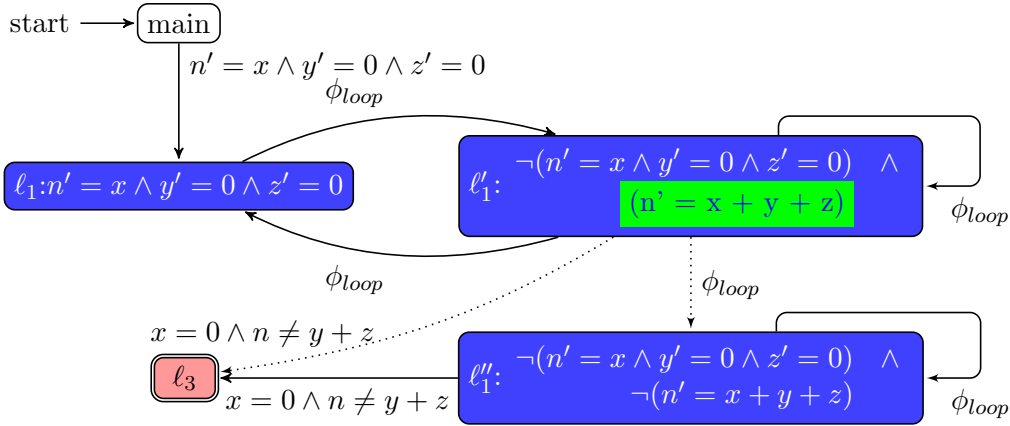


Figure 5: Final graph \mathcal{G} that proves the safety of program \mathcal{P} . The highlighted interpolant is an inductive invariant of the loop. The edges leading to subgraph (ℓ''_1, ℓ_3) are infeasible.

3 Preliminaries

A program is represented by a *program graph* $\mathcal{P} := (Loc, \ell_{init}, \ell_{err}, \delta)$ where Loc is a finite set of control locations, $\ell_{init} \in Loc$ is the initial location, $\ell_{err} \in Loc$ is the error location. The relation δ describes how control passes from one location to another and forms a directed graph. An edge $(\ell, \varphi, \ell') \in \delta$ is labeled with a *transition formula* φ . A transition formula is a formula over unprimed and primed program variables V and V' (see e.g., [13]). We think of a transition formula as representing a set of pairs of states (s, s') , s.t. $(s, s') \models \varphi$. For brevity of exposure, we assume that transition formulas are formulas over *all* unprimed and primed program variables.

This assumption is lifted in practice by using frame conditions.

A path in a program graph \mathcal{P} from a location ℓ_0 to a location ℓ_{n+1} is an alternating sequence of locations and transition formulas $\pi = \ell_0\varphi_0\ell_1\varphi_1\dots\ell_n\varphi_n\ell_{n+1}$ where $(\ell_i, \varphi_i, \ell_{i+1}) \in \delta$ for $0 \leq i \leq n$. A path from the initial location ℓ_{init} to the error location ℓ_{err} is called *error path*.

We extend the concept of transition formulas from edges to paths as follows: given a path $\pi = \ell_1\varphi_1\ell_2\varphi_2\ell_3$, where both φ_1 and φ_2 are formulas over the unprimed and primed variables $V = \{v_0, \dots, v_n\}$ and $V' = \{v'_0, \dots, v'_n\}$. The path formula $\varphi(\pi)$ is the *sequential composition* $\varphi_1 \circ \varphi_2$, such that

$$\exists v''_1, \dots, v''_n : \varphi_1[v''_0/v'_0 \dots v''_n/v'_n] \wedge \varphi_2[v''_0/v_0 \dots v''_n/v_n]$$

and $\varphi(\pi)$ is a formula over the unprimed and primed program variables.

Infeasibility and Interpolants. An error path in the program graph must not necessarily correspond to a real error. There may be no valuations for the program variables for which the transition formula is satisfied. We call paths that have an unsatisfiable transition formula infeasible.

Definition 1. A path $\pi = \ell_0\varphi_0\ell_1\dots\ell_n\varphi_n\ell_{n+1}$ in a program \mathcal{P} is infeasible if and only if its path formula $\varphi(\pi) := \varphi_0 \circ \dots \circ \varphi_n$ is unsatisfiable.

That is, the path π is infeasible if, for any valuation of the unprimed variables V , there is no valuation of V' , s.t. $\varphi(\pi)$ is satisfied. In particular, a location is unreachable if any path from ℓ_{init} to this location is infeasible. The program graph is safe if the error location ℓ_{err} is unreachable:

Definition 2. A program graph \mathcal{P} is safe if and only if every error path is infeasible.

For an infeasible error path we can compute Craig interpolants that separate the states reachable from the initial location from the states that can reach the error location on this path. We compute one interpolant for every location on the error path, using the following definition of interpolants for a path formula.

Definition 3. Given an unsatisfiable formula $\varphi_0 \circ \dots \circ \varphi_n$ where φ_i is a transition formula over V and V' , the sequence I_1, \dots, I_n of formulas over V is an inductive sequence of interpolants if the formulas

$$\varphi_0 \circ \neg I_1, \quad I_i \wedge \varphi_i \circ \neg I_{i+1} \text{ for } 1 \leq i < n, \quad I_n \wedge \varphi_n$$

are all unsatisfiable.

The I_i can be computed step by step as the Craig interpolant of the formulas $(\exists v_1 \dots v_n. I_{i-1} \wedge \varphi_{i-1})[v_1/v'_1 \dots v_n/v'_n]$ and $\varphi_i \circ \dots \circ \varphi_n$ (using $I_0 = true$). Note that the formulas contain only existential quantifiers provided that φ_i is quantifier free. Hence, the quantifiers can be removed by Skolemization and we can use interpolation algorithms for quantifier-free logics.

4 Program Graph Reduction

In this section, we will present a method to simplify the program graph without changing its correctness and without losing information about its structure. We will first sketch the method for loop free program graphs and then generalize it. Given a program graph without loops, there are only finitely many error paths whose infeasibility can be checked by a theorem prover call for every path. However, the number of paths may be exponential in the number of program transitions.

A better way to check correctness is to encode the branching structure in the formula by using disjunction. Given a location ℓ with outgoing edges $(\ell, \varphi_1, \ell_1), \dots, (\ell, \varphi_n, \ell_n)$ we can define its error transition formula describing all paths from ℓ to ℓ_{err} by

$$err \leftrightarrow (\varphi_1 \circ err_1) \vee \dots \vee (\varphi_n \circ err_n).$$

The symbols err_i for the other locations are similarly defined. This is only well defined for loop-free code; otherwise the definition would be cyclic. We can then check the satisfiability of err_{init} in conjunction using the above definition (the symbols err_i are boolean variables). This trick will move the burden of enumerating the error paths to the theorem prover. Moreover, the theorem prover can use its advanced techniques to avoid the exponential blow-up. Modern static checkers are based on this method [3].

For program graphs containing loops, one cannot encode the disjunction of the path formulas of all error path by a single (quantifier-free) transition formula. However, at least the loop-free fragments of the program graph can be transformed into a single transition formula. One way to achieve this is by large-block encoding [4]. The resulting program graph is much smaller and contains basically one location for every loop-header.

Besides being smaller, another advantage of large-block encoding is that only the program states at the beginning of a loop need to be considered in the model-checking process. Thus we concentrate on finding loop invariants instead of looking at every single computation step of the program. Moreover, the interpolating theorem prover looks at several parallel paths in the program at once. Thus it can output more informed interpolants that are more likely to capture the inductive invariant of the program, than if every path is considered separately.

We fold each loop-free subgraph in a program \mathcal{P} to a single edge using the fix-point application of sequential and disjunctive composition of edges. We express this using the two transformation rules given in Figure 6. The first rule compresses sequential code into a single edge by sequentially composing the edge labels. It is applicable if there is a location ℓ' with a single incoming edge. The transition formula of the incoming edge is composed with every transition formula on all outgoing edges to create new outgoing edges from the predecessor. The location ℓ' and all its incoming and outgoing edges are then removed.

Transformation 1. *Given a p graph $\mathcal{P} = (Loc, \ell_{init}, \ell_{err}, \delta)$. For any location $\ell' \in Loc \setminus \{\ell_{init}, \ell_{err}\}$, s.t. ℓ' has exactly one incoming edge $(\ell, \varphi, \ell') \in \delta$, we reduce the program graph \mathcal{P} as*

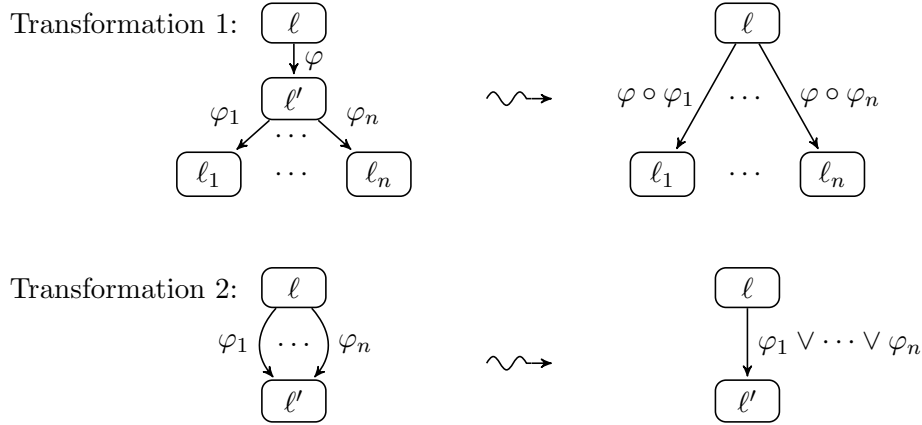


Figure 6: The reduction rules for simplifying the program graph. Our implementation of large-block encoding follows closely [4] (even though it was independently developed). Transformation 1 uses sequential composition to remove intermediate locations and Transformation 2 uses disjunction to remove multiple edges resulting from branches in the original program graph.

follows: 1) remove the location l' from Loc , and 2) replace all pairs of edges (l, φ, l') , (l', φ', l'') $\in \delta$ by a new edge $(l, \varphi \circ \varphi', l'')$.

Note that the rule duplicates the formula φ for every outgoing edge of location l' . We can avoid exponential blow-up by replacing φ by a boolean variable which is defined as φ , the same way we defined the variables err above.

After applying Transformation 1, each location in the program graph \mathcal{P} is either a sink or has more than one outgoing edge. The compression of sequential code by Transformation 1 can create multiple edges with the same source and destination location. We can fold such a sequence of edges into one by using the disjunctive composition.

Transformation 2. Given a program graph $\mathcal{P} = (Loc, l_{init}, l_{err}, \delta)$ and two nodes $l, l' \in Loc$. For any two edges (l, φ, l') , (l, φ', l') $\in \delta$, we reduce the graph by replacing these edges by a new edge $(l, \varphi \vee \varphi', l')$.

In a graph \mathcal{P} for which neither Transformation 1 nor Transformation 2 can be applied, we know, that each location is either sink node or a loop header. Hence, the application of Transformation 1 and Transformation 2 on a program \mathcal{P} does not change the satisfiability of the formula $\varphi(\mathcal{P})$.

Theorem 1. The fix-point application of Transformation 1 and Transformation 2 on a program graph $\mathcal{P} = (Loc, l_{init}, l_{err}, \delta)$ results in a program graph $\mathcal{P}_R = (Loc', l'_{init}, l'_{err}, \delta')$, where $Loc' \subseteq Loc$ and any location $l \in Loc'$ is reachable in \mathcal{P}_R if and only if l is reachable in \mathcal{P} . We say that \mathcal{P}_R is the reduced graph of \mathcal{P} .

A proof for Theorem 1 is given in [4].

5 Interpolant-based Model Checking

Our model checking algorithm given by Algorithm 1 takes a program graph \mathcal{P} as input and returns *safe*, if no error location can be reached, *unsafe*, if a feasible error path is found, or *unknown* if the algorithm can show neither of both. The algorithm first computes a reduced program graph \mathcal{P} using the transformation described in Section 4.

Our algorithm is based on abstraction refinement. The program graph represents the initial abstraction, which combines all program states that have the same location. The refinement steps will split these abstract states by formulas into those states that satisfy the formula and those that do not. The formulas are interpolants generated from an infeasible error path. After each split, there is a slicing step that removes all edges from the program graph that are no longer feasible.

Due to the splitting step, we will have several abstract states (we call them nodes) representing the same location. Each node will be associated with a formula (invariant), describing the set of concrete states represented by this node. Thus, the program graph is augmented by a labeling function Inv that labels each node ℓ with $Inv(\ell)$. Initially we start with the reduced program graph and $Inv(\ell)$ is *true* for every node. In the labeled program graph, a path is only feasible if there is a sequence of program variable valuations that satisfies the transition constraints and the invariant labeled to each state. Thus the path formula for a path $\pi = \ell_0\varphi_0\ell_1\dots\varphi_n\ell_{n+1}$ is augmented by the node invariants:

$$\varphi(\pi) := Inv(\ell_0) \wedge \varphi_0 \circ Inv(\ell_1) \wedge \varphi_1 \circ \dots \circ \varphi_n \circ Inv(\ell_{n+1}).$$

We define correctness for a labeled program graph exactly as before, i. e., the labeled program graph is safe if for all error paths π the path formula $\varphi(\pi)$ is unsatisfiable. It is obvious that the program graph is safe if and only if the labeled program graph with $Inv(\ell) = \textit{true}$ is safe.

The outer loop of the algorithm repeatedly checks if there exists an error path π in \mathcal{P} . If not, the algorithm terminates and returns that \mathcal{P} is safe. Otherwise we check whether π is feasible using the procedure `satisfiable`. This procedure checks the satisfiability of the path formula $\varphi(\pi)$. The procedure is implemented by an interpolating theorem prover (we use SMTInterpol¹). The prover returns *sat* if the error path π is feasible and then our algorithm returns *unsafe* because π is a counterexample that witnesses the reachability of the error location in \mathcal{P} . The prover returns *unsat* if the error path is infeasible, or *unknown* which will end the algorithm without result. In case of *unsat*, our algorithm computes a sequence of interpolants I_1, \dots, I_n for π using the procedure `Interpolants` (e.g., [14]). The procedure `Interpolants` returns one interpolant for each location ℓ_i on the path π . We use I_1, \dots, I_n to *split* the nodes into states that cannot reach the error location following the path π and states that cannot be reached from the initial location on π . The next step is called *slicing* and removes all edges (ℓ, φ, ℓ') that are not feasible *in every path* π because their transition constraint φ is incompatible with $Inv(\ell)$ and $Inv(\ell')$.

¹ <http://swt.informatik.uni-freiburg.de/research/tools/smtinterpol>

Algorithm 1: Model checker algorithm**Data:** $\mathcal{P} = (Loc, \ell_{init}, \ell_{err}, \delta)$;Map Inv from Loc to formulas;**Result:** $Safe$, $Unsafe$, or $Unknown$.

```

1 begin
2    $\mathcal{P} \leftarrow \text{Reduce}(\mathcal{P})$ ;
3   while exists an error path  $\pi$  in  $\mathcal{P}$  do
4     switch satisfiable( $\varphi(\pi)$ ) do
5       case sat: return unsafe;
6       case unsat:
7          $I_1, \dots, I_n := \text{Interpolants}(\pi)$ ;
8         foreach  $\ell_i$  in  $\pi$  do
9            $(\ell_i^{(1)}, \ell_i^{(2)}) \leftarrow \text{Split}(\ell_i, I_i)$ ;
10          Slice ( $\mathcal{P}$ );
11       otherwise return unknown;
12  return safe;

```

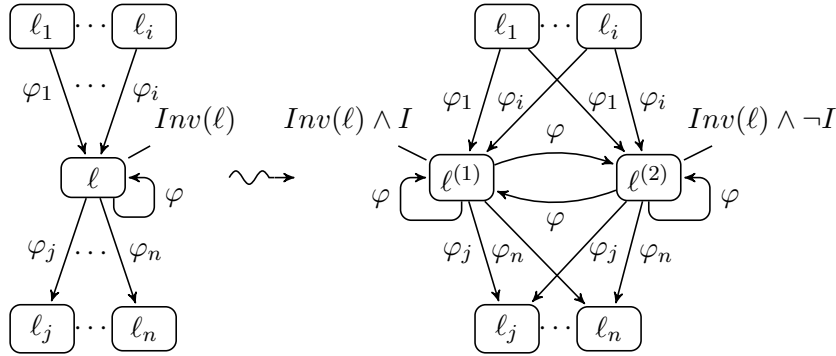


Figure 7: Splitting the node ℓ on the formula I in a labeled program graph. The node and its incoming and outgoing edges are duplicated and one copy of the node is labeled with I and the other with $\neg I$.

Splitting. The function $\text{Split}(\ell, I)$ in line 9 duplicates the node ℓ and augments the labeling of one copy with I and the labeling of the other copy with $\neg I$, see Fig. 7. When the node is duplicated, all incoming and all outgoing edges are duplicated as well (for the loop edge that is both incoming and outgoing we create four new edges).

Lemma 1. *Splitting a location does not change the set of feasible paths (modulo renaming of the nodes). The resulting program graph is correct if and only if the input graph is correct and it has the same feasible error paths.*

Proof. The second statement is a direct consequence of the first statement. To prove the first

statement, consider a feasible path of the original program graph visiting location ℓ once:

$$\pi = \ell_0 \varphi_0 \ell_1 \dots \ell \dots \varphi_n \ell_{n+1}.$$

Since π is feasible its path formula

$$\pi(\phi) = \text{Inv}(\ell_0) \wedge \varphi_0 \circ \dots \text{Inv}(\ell) \wedge \phi_i \circ \dots \varphi_n \circ \text{Inv}(\ell_{n+1})$$

is satisfiable. By definition of \circ , there exists a valuation of variables for location ℓ satisfying $\text{Inv}(\ell)$. Obviously this valuation must satisfy either I or $\neg I$. Hence for some $i \in \{1, 2\}$ the invariant $\text{Inv}(\ell^{(i)})$ is satisfied and the path

$$\pi = \ell_0 \varphi_0 \ell_1 \dots \ell^{(i)} \dots \varphi_n \ell_{n+1}$$

is feasible (with the same valuation). The argument can be inductively extended to paths visiting the location more than once (each time a different $\ell^{(i)}$ may be visited). \square

Slicing. In the slicing step the labeled program graph is simplified by removing infeasible edges. An edge (ℓ, φ, ℓ') is infeasible if the formula $\text{Inv}(\ell) \wedge \varphi \circ \text{Inv}(\ell')$ is unsatisfiable. Since this formula is a part of every path formula containing the edge, every path containing an infeasible edge is infeasible. Thus, removing the edge does not change the set of feasible paths. Removing edges may render subgraphs of the program graph unreachable. All unreachable edges and locations are also removed without affecting the feasible paths of the program.

Lemma 2. *The slicing operation preserves all feasible error paths in the labeled program graph and its correctness.*

Proof. As sketched above, slicing does not change the feasible paths and hence the feasible error paths. \square

Soundness and Progress. Using the above lemmas, we can immediately prove soundness of our algorithm:

Theorem 2 (Soundness). *The application of splitting and slicing on a program graph \mathcal{P} preserves all feasible error paths in \mathcal{P} . Hence, if the algorithm returns safe the original program graph has no feasible error path and if the algorithm returns unsafe the feasible error path found by the algorithm is also present in the original program graph.*

Proof. By Lemma 1 and Lemma 2. \square

Provided that the transition formulas φ in the program graph are from a decidable theory and that the interpolants are given in the same theory, the SMT solver will always terminate and return either *sat* or *unsat*. There are several decidable theories for which interpolation is possible,

e. g., quantifier free formulas over linear arithmetic and uninterpreted functions. For the theory of arrays there exist decidable fragments, however, it is not clear if there is a decidable fragment closed under interpolation.

If we use a decidable and interpolating theory, our algorithm will never terminate with *unknown*. However, since the software model checking problem is undecidable (even for simple integer programs using only linear arithmetic), we cannot prove that our algorithm will always terminate. However, we can show a progress property:

Theorem 3 (Progress). *In each loop iteration our algorithm will exclude one infeasible error path from the program.*

Proof. Let I_1, \dots, I_n be the interpolants for the infeasible error path $\pi = \ell_{init}\varphi_0 \dots \varphi_n\ell_{err}$. By the definition of interpolants we know that the formulas

$$Inv(\ell_{init}) \wedge \varphi_0 \circ \neg I_1, \quad I_i \wedge Inv(\ell_i) \wedge \varphi_i \circ \neg I_{i+1}, \quad I_n \wedge Inv(\ell_n) \wedge \varphi_n \circ Inv(\ell_{err})$$

are unsatisfiable. After splitting the nodes ℓ_1, \dots, ℓ_n , the edges from ℓ_{init} to $\ell_1^{(2)}$, from $\ell_i^{(1)}$ to $\ell_{i+1}^{(2)}$, and from $\ell_n^{(1)}$ to ℓ_{err} are infeasible and thus removed in the slicing step. Thus after slicing, the nodes $\ell_i^{(2)}$ and ℓ_{err} are not reachable *on the path* π . This shows that error path π is not present in the resulting program graph any more. \square

6 Experimental Evaluation

We have implemented our algorithm in a tool called *Ultimate*. The implementation is in Java and takes Boogie programs [11] as input. We evaluate our model checker on a set of benchmark programs. We compare the presented algorithm with the existing algorithms Impact [1], Blast [5], and Wolverine [10] in terms of how often a result can be found. In our experiments we use Java implementation of Impact that takes Boogie programs as input. For Impact we use our own implementation of McMillan’s algorithm, optionally with the large-block encoding pre-processing step. This allows us to directly evaluate the benefits of the different algorithms. To compare our implementation with third party implementations we use the binary distributions of Blast ² and Wolverine ³. Note that Blast and Wolverine check C programs and thus, their results are not directly comparable with our implementation. E.g., in Boogie, the type `int` is by definition unbounded. For C programs this type is usually bounded and some model checker check for integer overflow and flag them as errors. Thus, the result may differ between different model checkers. As our focus is on how often the tools can find a result rather than on the concrete result, we still believe it is worth comparing them.

²<http://mtc.epfl.ch/software-tools/blast/index-epfl.php>

³<http://www.cprover.org/wolverine/>

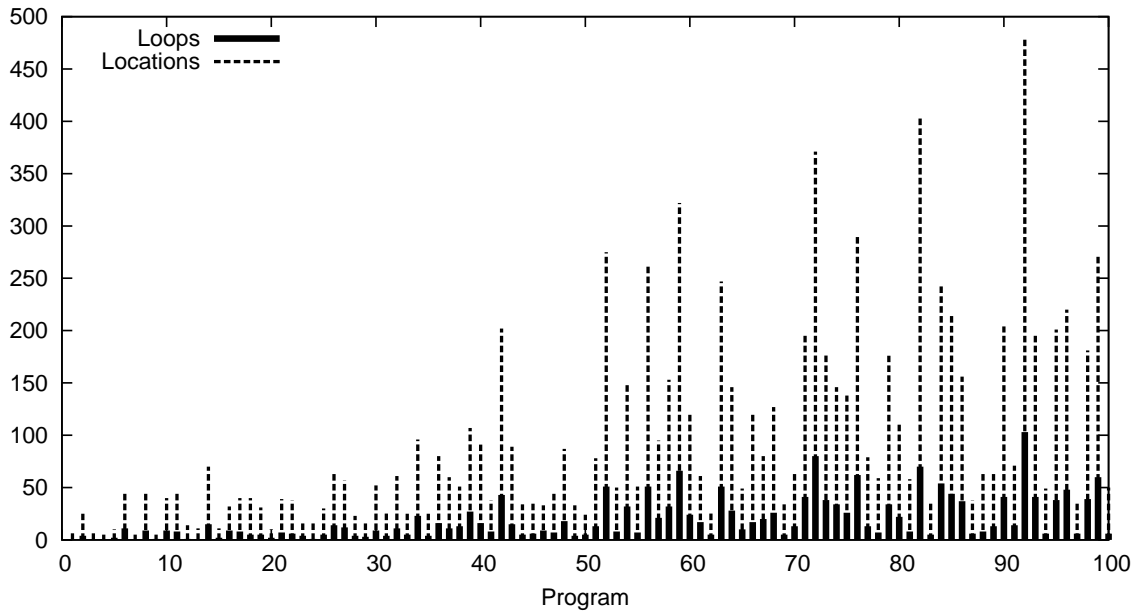


Figure 8: Total number of locations and loops in each randomly generated program.

Experimental Material. We conduct our experiments on two different data sets. We use 13 hand written example programs and a set of 100 random generated programs. The generated programs vary in size and complexity. Each random program consists of one method with 5 local variables. The body of the method is a sequence of up to 15 control-flow statements; each statement is either an `if-then-else` statement or a `while` loop. Each control-flow statement has up to 5 sub-statements. The maximal nesting depth is 5. At the end of the generated method we add a sequence of randomly generated assertions. For the comparison with Blast and Wolverine, we use equivalent C programs (As these programs only modify integers, the translation from Boogie to C is straight forward). Fig. 8 gives an overview of the number of locations and loops of the generated files. Our implementation and the experimental material is publicly available⁴. All experiments were repeated several times on a standard laptop computer with 2 GB of RAM. For each file, we killed the model checking process after 10 minutes and report a timeout.

Comparison of the Algorithms. Table 1 shows the results of our hand-written experiments. The benchmark programs are partly taken from the Blast benchmarks and partly hand-written. The results show that our approach is fast but, more importantly, that it is able to detect an

⁴<http://www.informatik.uni-freiburg.de/~ermis/ictac11.zip>

Benchmark		Ultimate			Impact			Blast 2.5		Wolverine	
	Nodes	Splits	Time	Result	Splits	Time	Result	Time	Result	Time	Result
1	14	0	21	safe	19	20	safe	104	unsafe	261	safe
2	5	0	43	safe	11	69	safe	0	na	128	safe
3	7	7	175	safe	0	0	na	32	safe	0	na
4	7	7	166	safe	0	0	na	20	safe	0	na
5	36	18	987	safe	0	0	na	28	safe	877	safe
6	31	1	181	unsafe	0	0	na	44	unsafe	528	safe
7	11	0	4	unsafe	13	22	unsafe	108	unsafe	2889	unsafe
8	17	6	413	safe	0	0	na	0	na	0	na
9	11	0	55	safe	0	0	na	0	na	4774	safe
10	26	0	34	safe	1607	976	safe	76	safe	72	safe
11	50	0	47	safe	0	0	na	400	safe	203	safe
12	98	0	51	safe	0	0	na	0	na	687	safe
13	194	0	65	safe	0	0	na	0	na	2913	safe

Table 1: Results on the hand-written benchmarks. The columns indicate the number of locations in the program graph of each benchmark and the results by the respective tool in terms of splits (loop unwinding), computation time and verification outcome.

inductive loop invariant for *all* programs. This supports our claim that computing the sequence of interpolants for all paths passing the same loops using a single call to the interpolation generator increases the possibility to identify an inductive loop invariant.

We further compare our implementation with existing tools on the randomly generated programs. Table 2 shows the results of the different algorithms. First, we have to understand that Wolverine and Blast operate on C files and have different results for particular files. This is due to the fact that these tools assume the ANSI-C type system (e.g., limited integer range) while our implementation of Ultimate and Impact operate on Boogie programs where numeric types are unbounded. This is a threat to validity, but as the purpose of this experiment is to compare how often these approaches can come to a result we assume that this is still a valid experiment. We can see that Ultimate comes to a result for 76 of 100 programs and thus terminates more often than other approaches. Fig. 9 compares the different tools on the random benchmarks in more

	Ultimate	Impact	Impact+LBE	Blast 2.5	Wolverine
UNSAFE	47	33	44	25	60
SAFE	29	18	26	11	3
UNKNOWN	24	49	30	64	37

Table 2: Results of the different model checking algorithms on the random generated programs

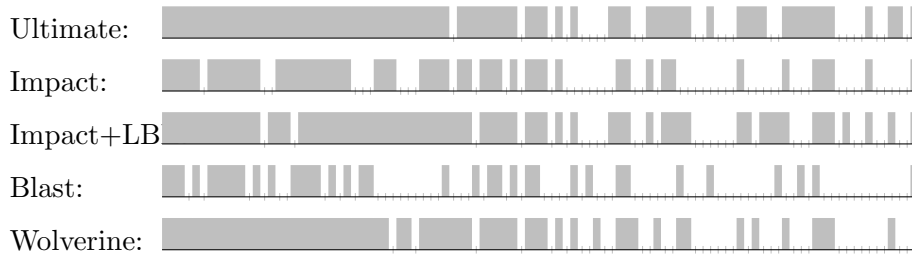


Figure 9: Results for the different model checking algorithms on the random generated programs. the x-axis ranges over the number of programs, an impulse on the y-axis represents a result different from *Unknown*.

detail. A positive value on the y-axis represents that the tool was able to compute a result. The figure shows that Ultimate can come to a solution significantly more often than Impact with out large block encoding or Blast. For larger programs, Ultimate also has a higher chance to achieve a result than Wolverine. Impact with large-block encoding is significantly better than without and its performance is similar to Ultimate.

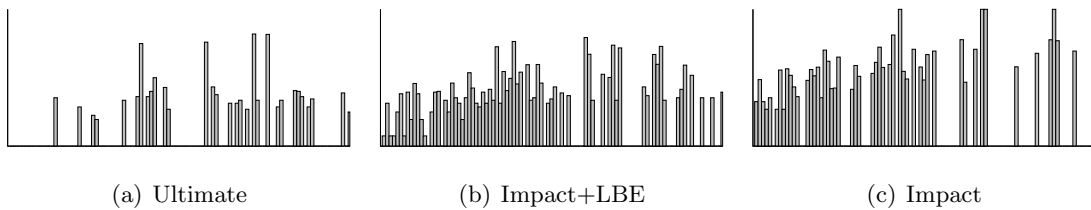


Figure 10: Total number refinement steps when analyzing a program for Ultimate and Impact. The x-axis ranges over the number of program, the y-axis from 1 to 5000 on a logarithmic scale.

Due to the incomparability of *C* and Boogie programs we only discuss the differences between Ultimate and Impact in more detail. Fig. 10 shows how often Ultimate, Impact+LBE, and Impact have to refine the abstraction. Note that the y axis is logarithmic. While Ultimate needs at most 1000 refinement steps for complex programs, Impact needs up to 7000 refinements steps for the same example. The experiments show that, at any point, Ultimate needs less refinement steps than Impact with large-block encoding, which needs less refinement steps than Impact. Note that timeout of the model checker results in a missing bar.

In Fig. 11 we compare the total computation time per program for Ultimate and Impact. The experiments show that Ultimate is faster on all benchmark programs. Even though, the speed-up is not always significant, the combination of improved computation time and the fact that Ultimate is able to find a result for more programs than Impact lets us conclude that the presented algorithm is an overall improvement.

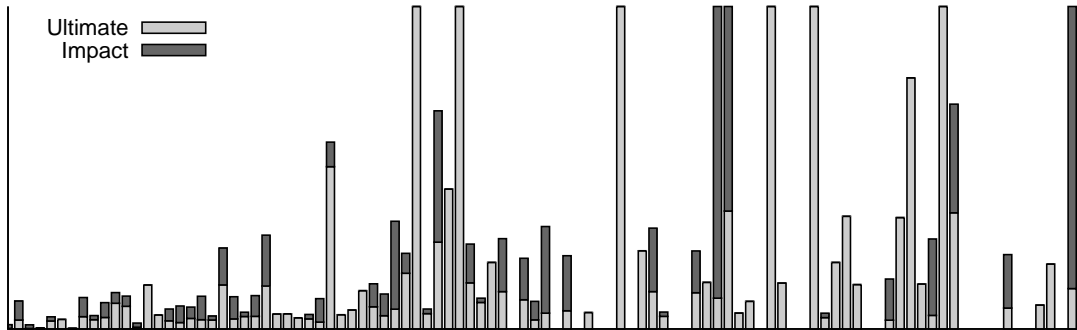


Figure 11: Total computation time per file for Ultimate and Impact. The x-axis ranges over the number of program, the y-axis from 0 to 1000sec. For timeouts in inconclusive results we set the time to 0.

Threats to Validity. The main internal threat to validity arises from the used theorem prover. If it is possible to derive an inductive invariant depends solely on the interpolants computed by the used theorem prover. We believe that our approach to compute one interpolant for all error paths that pass a particular location at the same time allows any theorem prover to infer more suitable interpolants. However, comparing the interpolants computed by different theorem provers remains part of our future work.

An external threat to validity is the use of the Boogie language. Boogie is a simple intermediate verification language. We cannot predict how our method will be able to deal with real programs translated into Boogie or how an appropriate translation has to loop like. This, in particular, may affect the comparison with Wolverine and Blast. Even though the Boogie and C programs used in this experiments are equivalent, Blast and Wolverine may assume different C semantics which may result in a more expensive computation.

7 Related Work

We compare our tool to modern software model checkers like Blast [5], CPAChecker [4], and IMPACT [14]. The most common approaches, e.g. Blast, SLAM [2], SATABS [7], use CEGAR algorithms with predicate abstraction. In predicate abstraction the model is refined by newly

obtained predicates e.g. derived from interpolants. In our approach we use the interpolants themselves in order to refine the model. This technique has been introduced by Ken McMillan [12]. In [14] he showed that his implementation of an interpolation-based model checker, IMPACT, has a serious performance gain compared to tools using predicate abstraction by avoiding the abstract image computation. As abstraction process IMPACT uses lazy abstraction as known from BLAST. The software model checking procedure of IMPACT has also been implemented in other tools e.g. Wolverine⁵. The main difference to our approach is the abstraction technique. Our approach uses slicing abstraction[ref]. This abstraction process was implemented in a tool called SLAB [8]. But in contrast to our approach, SLAB uses predicate abstraction. In order to adapt interpolation to slicing abstraction we use a technique called large block encoding [4] which is implemented in CPAChecker. CPAChecker uses the same software model checker procedure as Blast but compresses the model by joining sequential code segments to single transitions.

8 Conclusion

We have presented a new interpolant-based model checking algorithm based on the idea of slicing abstraction. The algorithm uses large block encoding to first, limit the number of traces exercised by the model checker, and second to give the theorem prover more information about the program when inferring interpolants. Our experimental evaluation indicates that the combination of both techniques yields good results in terms of computation time and detection rate compared to other tools. In particular our algorithm was able to return a result more often than any other model checking algorithm in our experiments. We are aware that some of the experimental results are not directly comparable, on the other hand the results show the tendency that our algorithm can outperform existing model checkers.

In our future work we investigate alternative ways to compute the interpolants for a particular location. While large-block encoding has shown to be useful in our experiments, it is possible to encode more through one location into a formula. We will analyze how the size of the formula affects the termination behavior of our algorithm and the computation time of each theorem prover query. We have seen, that reducing subgraphs into a single transition formula helps us to improve speed and precision. However, in general, the complexity of the resulting formula alters the response behavior of the theorem prover. We will experiment with different strategies to reduce the program graph and compare their efficiency in terms of total theorem prover calls and computation time.

We will investigate how the proposed algorithm can be applied efficiently to C and Java program to allow a more detailed comparison with other model checking tools in terms of performance and detection rate. We will investigate how the choice of other interpolating decision procedures (e.g., [10]) affects the ability of our algorithm to detect loop invariants.

We conclude that it is possible for slicing abstraction to compute the abstract post from path

⁵<http://www.cprover.org/wolverine/>

formulas instead of predicates. We have seen that the presented algorithm terminates more often than lazy abstraction based approaches. However, more research and evaluation is needed to evaluate and improve the presented algorithm.

Acknowledgements. This work is partly supported by the project ARV funded by Macau Science and Technology Development Fund.

References

- [1] N. Amla and K. L. McMillan. Combining abstraction refinement and sat-based model checking. In *TACAS*, pages 405–419, 2007.
- [2] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, New York, NY, USA, 2002. ACM.
- [3] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31:82–87, September 2005.
- [4] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, 2009.
- [5] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9:505–525, October 2007.
- [6] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In F. Arbab and M. Sirjani, editors, *FSEN*, 2007.
- [7] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.
- [8] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. Slab: A certifying model checker for infinite-state concurrent systems.
- [9] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, New York, NY, USA, 2002. ACM.
- [10] D. Kroening and G. Weissenbacher. An interpolating decision procedure for transitive relations with uninterpreted functions. In *Haifa Verification Conference*, pages 150–168, 2009.
- [11] K. R. M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at <http://research.microsoft.com/~leino/papers.html>.
- [12] K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.

-
- [13] K. L. McMillan. Applications of craig interpolants in model checking. In *TACAS*, pages 1–12, 2005.
- [14] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.