Splitting via Interpolants*

Evren Ermis, Jochen Hoenicke, and Andreas Podelski

University of Freiburg
{ermis,hoenicke,podelski}@informatik.uni-freiburg.de

Abstract. A common problem in software model checking is the automatic computation of accurate loop invariants. Loop invariants can be derived from interpolants for every path leading through the corresponding loop header. However, in practice, the consideration of single paths often leads to very path specific interpolants. Inductive invariants can only be derived after several iterations by also taking previous interpolants into account.

In this paper, we introduce a software model checking approach that uses the concept of *path insensitive* interpolation to compute loop invariants. In contrast to current approaches, path insensitive interpolation summarizes *several* paths through a program location instead of one. As a consequence, it takes the abstraction refinement considerably less effort to obtain an adequate interpolant. First experiments show the potential of our approach.

1 Introduction

In software model checking, abstraction refinement is used to prove properties of a system on an abstract model without actually expanding this model to the state level. 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. This can be achieved by extending the model with computed invariants. The use of Craig interpolants is one promising approach for this purpose [15]. However, interpolation on single paths computes path specific interpolants. In order to find an accurate invariant the desired interpolant must be inductive. Current interpolation-based approaches find an interpolant for each infeasible error path separately. These interpolants can be combined into a single inductive invariant for all paths.

We introduce the concept of *path insensitive interpolation* as a technique to derive inductive interpolants more directly. The idea is to put more information into the interpolation process by considering several paths through the observed location. Thus, we are more likely to obtain an inductive invariant for this location. We present a novel software model checking approach that combines *path insensitive interpolation* with splitting as abstraction refinement.

* A preliminary version appeared as UNU-IIST, Macau, Technical Report 449, June 2011

This is the author's version of the work published in *Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, LNCS 7148, pages 186–201. Springer, 2012. The orginal publication is available at www.springerlink.com/index/10.1007/978-3-642-27940-9_13

Splitting separates the states along an infeasible path to those reachable from the initial location and those leading to the erroneous location. The refined model can contain several nodes representing the same location of the original program. Each node carries an invariant that characterizes a set of states. Using splitting as refinement step has the benefit, that the loop invariant does not have to be derived as a single inductive interpolant. It can be constructed as a union of all interpolants. The main task is to find useful interpolants.

Path insensitive interpolation returns an interpolant for a location ℓ considering several paths through ℓ. If in the extreme case all possible paths through ℓ are considered, the interpolant is guaranteed to be the right inductive invariant for ℓ. If loops are present this is not possible, but one can still merge loop-free paths through ℓ to get an interpolant that holds for all the considered paths. This interpolant will be an inductive invariant for this location ℓ for the loop-free subprogram considered. In a first approach our algorithm collapses non-looping subprograms into single transitions using *large-block encoding* (LBE) [4]. In the resulting graph, each path corresponds to a set of original program paths. On this compressed model we can compute path interpolants [16]. Thus, we can derive interpolants from the refutation of sets of paths rather than single paths. This approach allows to compute interpolants that are path insensitive modulo the loop iterations of the program. E.g. programs with multiple outer loops cannot be compressed by LBE to achieve path insensitive interpolation but our results show that *partial path insensitivity* still works efficiently. It helps to reduce the number of splits needed to an extent that our algorithm can efficiently handle programs of a realistic size.

In the following, we illustrate our model checking approach using an example in Section 2. In Section 3 we introduce the basic definitions for technical Section 4. In Section 4 we present our interpolation-based model checking approach. An experimental evaluation of the approach is given in Section 5.

2 Example

We will illustrate the approach by applying it to program `main` (Fig. 1). The program has non-deterministic branches. The sum of x, y, and z is equal to the initial value n in each iteration of the loop. Consequently, the assertion $n = y + z$ holds when the loop exits with $x = 0$. The program has the corresponding program graph $\mathcal{P}$ (Fig. 2).

Our approach uses splitting as refinement step. The path interpolants [16] derived from infeasible error paths are used as splitting criteria. An error path is encoded as a FOL formula and passed to the interpolating SMT solver. Our approach compresses paths in the model by applying large block encoding (LBE). LBE compresses loop-free subgraphs to single edges. Hence, checking one path in the compressed model covers multiple paths in the original program graph. The effect is that the obtained interpolants are at least partially path insensitive. LBE iteratively (1) compresses *sequential nodes* to single edges by using conjunctions and (2) merges *multiple edges* by using disjunctions. Via the intro-

```
1  procedure main() {
2          var x,y,z,n: int;
3          assume(n == x && y == 0 && z == 0);
4          while(x != 0) {
5                  if (*) {
6                          x := x + 1;
7                          y := y − 1;
8                  }
9                  if (*) {
10                         y := y + 1;
11                         z := z − 1;
12                 }
13                 if (*) {
14                         x := x − 1;
15                         z := z + 1;
16                 }
17         }
18         assert(n == y + z);
19 }
```

Fig. 1. Code of program `main`. Non-deterministically increments one variable whilst decrementing a second variable. The program is safe if the assertion (Line 18) holds on every execution.

duced disjunctions the decision of the branching is shifted to the interpolating SMT solver. The edges of the compressed model represent contiguous loop-free code segments.

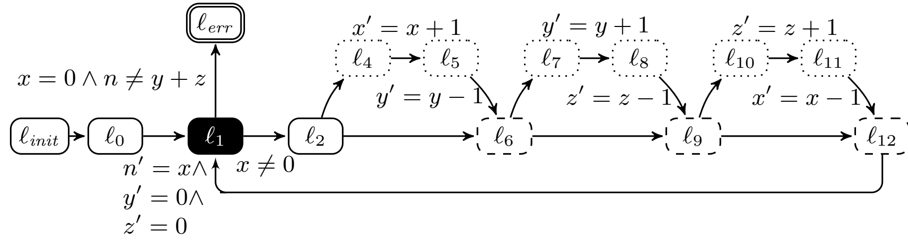
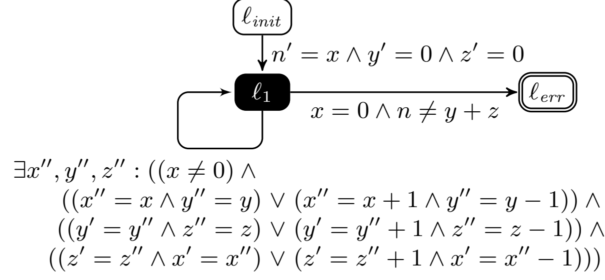


Fig. 2. Program graph P of `main` (Fig. 1). Edges without labeling carry the formula $\top$. ℓ_{err}'s guard is the negated assertion (Fig. 1, Line 18). ℓ_1 represents the loop head (Fig. 1, Line 4). ℓ_6, ℓ_9 and ℓ_{12} are nodes that will have entering multiple edges because of the preceding branches.



$$\begin{array}{l}\exists x'', y'', z'' : ((x \neq 0) \wedge \\ \quad ((x'' = x \wedge y'' = y) \vee (x'' = x + 1 \wedge y'' = y - 1)) \wedge \\ \quad ((y' = y'' \wedge z'' = z) \vee (y' = y'' + 1 \wedge z'' = z - 1)) \wedge \\ \quad ((z' = z'' \wedge x' = x'') \vee (z' = z'' + 1 \wedge x' = x'' - 1)))\end{array}$$

Fig. 3. The resulting model after compressing P (Fig. 2). The entire body of the loop is encoded in a single edge. Only the loop header (Fig. 1, Line 4) cannot be reduced any further.

Sequential nodes ℓ_i and ℓ_{i+1} are compressed, if ℓ_i is the only predecessor of ℓ_{i+1} and they're connected by a single transition (Dotted nodes (Fig. 2)). E. g., $x' = x + 1 \wedge y' = y - 1$ encodes the `then`-branch of the conditional branching in line 5 (Fig. 1). If a variable is changed in both transition, we introduce new auxiliary variable (e. g., y'') for the intermediate value. An alternative would be to use single static assignment (SSA).

Multiple edges occur if the original program has conditional branchings. The branchings are merged by joining the formulas with a disjunction. If the branches disagree on the changed variable they have to be adapted to each other by inserting frame conditions, e. g., the first `else`-branch from ℓ_2 to ℓ_6 is changed to $x' = x \wedge y' = y$. In our example (Fig. 3) the disjunctions encode the three conditional branchings at Line 5, 9, and 13 (Fig. 1). LBE provides a partially path insensitive observation of locations and their interpolants. In Figure 3 the shortest error path through ℓ_1 has the corresponding FOL formula

$$\begin{array}{ll}\ell_{init} \to \ell_1 & \exists x, y''', z''', n' : ((n' = x \wedge y''' = 0 \wedge z''' = 0) \wedge \\ \ell_1 \to \ell_1 & \left(\begin{array}{l}\exists x'', y'', z'' : ((x \neq 0) \wedge \\ \quad ((x'' = x \wedge y'' = y''') \vee (x'' = x + 1 \wedge y'' = y''' - 1)) \wedge \\ \quad ((y' = y'' \wedge z'' = z''') \vee (y' = y'' + 1 \wedge z'' = z''' - 1)) \wedge \\ \quad ((z' = z'' \wedge x' = x'') \vee (z' = z'' + 1 \wedge x' = x'' - 1)))\end{array}\right) \wedge \\ \ell_1 \to \ell_{err} & (x' = 0 \wedge n' \neq y' + z'))\end{array}$$

The solver returns either a configuration that proves the feasibility of the error path or an array of interpolants [16]. Each interpolant corresponds to a location but is not bound to a single execution path. It rather summarizes a set of execution paths through this location. This way we get partial path insensitivity. It is partial because it summarizes all paths through ℓ_1 modulo the loop iterations. If the path is feasible, the program is unsafe. If it is not feasible, the interpolants are used to split the path (Fig. 4). The returned interpolant I_i (e.g. $(n = x) \wedge (y = 0) \wedge (z = 0)$, Fig. 4) is appended to the corresponding node $\ell_{i+1}(\ell_1$, Fig. 4). Its negation is appended to the split node ℓ'_{i+1}. ℓ'_{i+1} inherits

all incoming and outgoing edges of ℓ_{i+1}. Infeasible edges of ℓ_{i+1} and ℓ'_{i+1} are removed from the model (dotted edges, Fig. 4).

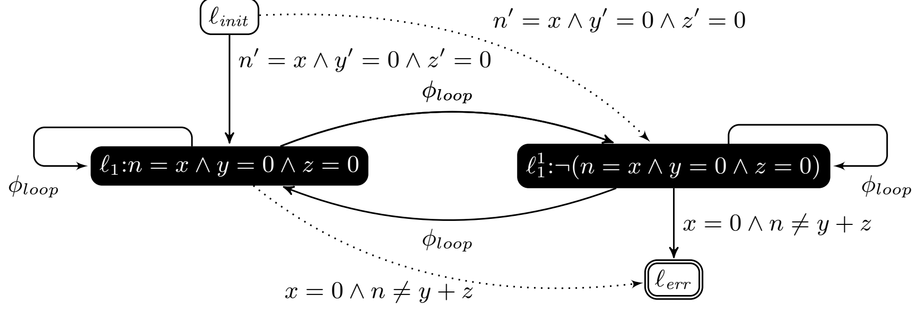


Fig. 4. Model after splitting the error path. The control flow is partitioned by appending the interpolant $n = x \wedge y = 0 \wedge z = 0$ to ℓ_1 and its negation to ℓ_1^1. The label ϕ_{loop} denotes the formula on the loop edge of Figure 3. Dotted edges are infeasible and will be removed.

In the next iteration we take the error path $\ell_{init}, \ell_1, \ell_1^1, \ell_{err}$. The edge from ℓ_1 to ℓ_1^1 is annotated with the disjunction from the edge ℓ_1 to ℓ_1 in the previous graph. Due to this disjunction the interpolant generator has to find an interpolant that works for all branches through the if statements. This will most probably result in the interpolant $n = x + y + z$, which is then used to split ℓ_1^1 (Fig. 5). Node ℓ_1 is not split, since its interpolants is **true**. The subsequent feasibility check of the edges renders the subgraph, consisting of ℓ_1^2 and ℓ_{err}, unreachable from ℓ_{init}. Hence after removing the infeasible edges, our model does not contain any error paths. The algorithm stops and has proven the safety of the program `main` by deriving the loop invariant.

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., [15]). 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 introducing a frame condition like $x = x'$ only when necessary, i. e., when computing the disjunction with a formula that changes x. 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 $\pi = \ell_0\varphi_0\ell_1\varphi_1 \ldots \ell_n\varphi_n\ell_{n+1}$ where $(\ell_i, \varphi_i, \ell_{i+1}) \in \delta$ for $0 \leq i \leq n$.

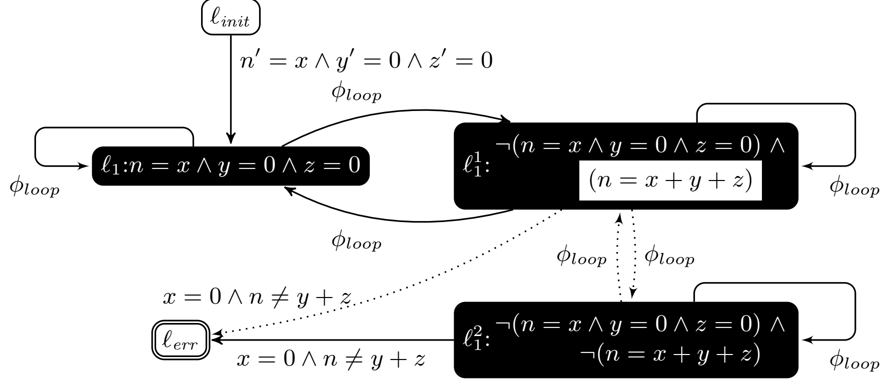


Fig. 5. Final model that proves the safety of program `main`. The highlighted interpolant is a loop invariant. The edges leading to subgraph (ℓ_1^2,ℓ_{err}) are infeasible.

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 $V = \{v_0, \ldots, v_n\}$ and $V' = \{v'_0, \ldots, v'_n\}$. The path formula $\varphi(\pi)$ is the *sequential composition* $\varphi_1 \circ \varphi_2$, such that

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

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 $\pi = \ell_0\varphi_0\ell_1 \ldots \ell_n\varphi_n\ell_{n+1}$ in a program $\mathcal{P}$ is* infeasible *if and only if its path formula $\varphi(\pi) := \varphi_0 \circ \ldots \circ \varphi_n$ 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 $\varphi_0 \circ \cdots \circ \varphi_n$ where φ_i is a transition formula over V and V', the sequence $I_1, \ldots, I_n$ of formulas over V is an* inductive sequence of interpolants *if the formulas*

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

are all unsatisfiable.

The I_i can be computed step by step as the Craig interpolant of the formulas $(\exists v_1 \ldots v_n . I_{i-1} \wedge \varphi_{i-1})[v_1/v_1' \ldots v_n/v_n']$ and $\varphi_i \circ \cdots \circ \varphi_n$ (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 Splitting via Path Insensitive Interpolants

4.1 Underlying Splitting Algorithm

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.

Our algorithm is based on abstraction refinement. An abstract state of the program is a tuple (ℓ, Inv) where ℓ is a program location and Inv a formula over the program variables. It represents the concrete states of the program where the program counter is in location ℓ and program variables fulfill the formula Inv. The initial abstraction is given by the program graph where each node is additionally labeled with the invariant **true** represents the initial abstraction. I. e., all program states that have the same location are combined into one abstract state. A refinement step splits an abstract state by a formula into those states that satisfy the formula and those that do not. The formulas are interpolants computed 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 associated with a different formula (invariant). In the abstract transition system, a path is feasible if there is a sequence of program variable valuations that satisfies the transition constraints and the invariants labeled to each state. Thus the path formula for a path $\pi = (\ell_0, Inv_0)\varphi_0(\ell_1, Inv_1) \ldots \varphi_n(\ell_{n+1}, Inv_{n+1})$ is augmented by the node invariants:

$$\varphi(\pi) := Inv_0 \wedge \varphi_0 \circ Inv_1 \wedge \varphi_1 \circ \ldots \circ \varphi_n \circ Inv_{n+1} \,.$$

We define correctness for a abstract transition systems exactly as for program graphs, 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 initial abstract transition system is safe where each location ℓ is replaced with the node $(\ell, \mathbf{true})$.

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. If the prover determines that the formula is satisfiable, i. e., the error path π is feasible, our algorithm returns *unsafe* because π is a counterexample that witnesses the reachability of the error location in $\mathcal{P}$. Otherwise the error path is infeasible and our algorithm computes a sequence of interpolants $I_1, \ldots, I_n$ for π using the procedure `Interpolants` (e.g.,[16]). The procedure `Interpolants` returns one interpolant for each location ℓ_i on the path π. We use $I_1, \ldots, I_n$ 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 $((\ell, Inv), \varphi, (\ell', Inv'))$ that are not feasible *in every path* π because their transition constraint φ is incompatible with Inv and Inv'.

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    foreach ℓ_i in P do
3      Replace ℓ_i with (ℓ_i, true)
4    while exists an error path π in P do
5      switch satisfiable(φ(π)) do
6        case sat: return unsafe;
7        case unsat:
8          I_1, ..., I_n := Interpolants(π);
9          foreach (ℓ_i, Inv_i) in π do
10           Split (ℓ_i, Inv_i) into (ℓ_i, Inv_i ∧ I_i), (ℓ_i, Inv_i ∧ ¬I_i);
11         Slice (P);
12       otherwise return unknown;
13   return safe;
```

Splitting. The function `Split` in line 10 duplicates the node (ℓ, Inv) and augments the labeling of one copy with I and the labeling of the other copy with $\neg I$, see Fig. 6. 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 (except for annotating a different invariant). The resulting abstract transition system is*

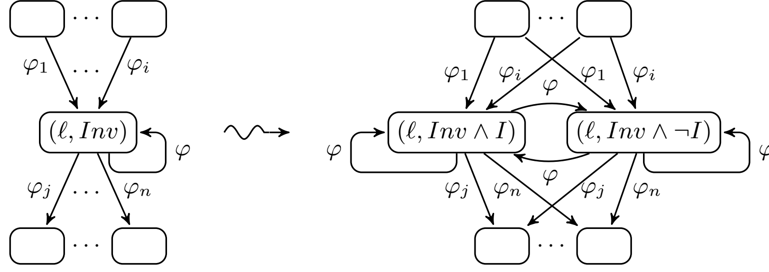


Fig. 6. Splitting the node ℓ on the formula I in a labeled program graph. The node and its incoming and outgoing edges are duplicated and one copy of the node is labeled with I and the other with $\neg I$.

correct if and only if the input system 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:

$$\pi = (\ell_0, Inv_0)\varphi_0(\ell_1, Inv_1) \ldots (\ell, Inv) \ldots \varphi_n(\ell_{n+1}, Inv_{n+1}).$$

Since π is feasible its path formula

$$\pi(\phi) = Inv_0 \wedge \varphi_0 \circ \ldots Inv \wedge \phi_i \circ \ldots \varphi_n \circ Inv_{n+1}$$

is satisfiable. By definition of $\circ$, there exists a valuation of variables for location ℓ satisfying Inv. Obviously this valuation must satisfy either I or $\neg I$. Hence either $F = I$ or $F = \neg I$ is satisfied and the path

$$\pi = (\ell_0, Inv_0)\varphi_0(\ell_1, Inv_1) \ldots (\ell, Inv \wedge F) \ldots \varphi_n(\ell_{n+1}, Inv_{n+1})$$

is feasible (with the same valuation). The argument can be inductively extended to paths visiting the location more than once (each time a different $(\ell, Inv \wedge F)$ may be visited). □

Slicing. In the slicing step the labeled program graph is simplified by removing infeasible edges. An edge $((\ell, Inv), \varphi, (\ell', Inv'))$ is infeasible if the formula $Inv \wedge \varphi \circ Inv'$ 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 abstract transition system and its correctness.*

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

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

Theorem 1 (Soundness). *The application of splitting and slicing on a program graph $\mathcal{P}$ as performed by Algorithm 1 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.

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, e. g., [7]. Recently there has also been work on a decidable fragment closed under interpolation [9].

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

Theorem 2 (Progress). *In each loop iteration our algorithm will exclude one infeasible error path from the program.*

Proof. Let $I_1, \ldots, I_n$ be the interpolants for the infeasible error path

$$\pi = (\ell_{init}, Inv_{init})\varphi_0(\ell_1, Inv_1)\ldots\varphi_n(\ell_{err}, Inv_{err}).$$

By the definition of interpolants we know that the formulas

$$Inv_{init} \wedge \varphi_0 \circ \neg I_1,\;\; I_i \wedge Inv_i \wedge \varphi_i \circ \neg I_{i+1},\;\; I_n \wedge Inv_n \wedge \varphi_n \circ Inv_{err}$$

are unsatisfiable. After splitting, the edge φ_0 from $(\ell_{init}, Inv_{init})$ to $(\ell_1, Inv_1 \wedge \neg I_1)$, the edges φ_i from $(\ell_i, Inv_i \wedge I_i)$ to $(\ell_{i+1}, Inv_{i+1} \wedge \neg I_{i+1})$ $(1 \leq i < n)$, and the edge φ_n from $(\ell_n Inv_n \wedge I_n)$ to (ℓ_{err}, Inv_{err}) are infeasible and thus removed in the slicing step. Thus after slicing, the nodes $(\ell_i, Inv_i \wedge \neg I_i)$ and (ℓ_{err}, Inv_{err}) are not reachable *on the path* π. This shows that error path π is not present in the resulting program graph any more. □

4.2 Path Insensitive Interpolation

In this section, we will present a first approach to apply path insensitive interpolation. Path insensitive interpolation finds an interpolant I for a location ℓ that holds for any infeasible error path $\pi = (\ell_{init}, Inv_{init})\ldots(\ell, Inv)\ldots(\ell_{err}, Inv_{err})$. Therefore the interpolant I is not just a summary in the context of a single path but *insensitively* of any path.

Definition 4. *Given a program graph $\mathcal{P} = (Loc, \ell_{init}, \ell_{err}, \delta)$. For any location $\ell \in Loc \setminus \{\ell_{init}, \ell_{err}, \}$, there exists a set Π_ℓ of all error paths*

$$\pi_i = (\ell_{init}, Inv_{init}) \dots (\ell^i_{n-1}, Inv^i_{n-1}), (\ell, Inv), (\ell^i_{n+1}, Inv^i_{n+1}) \dots (\ell_{err}, Inv_{err}).$$

If there exists an interpolant I, s.t.

$$\varphi((\ell_{init}, Inv_{init}) \dots (\ell^i_{n-1}, Inv^i_{n-1})) \Rightarrow I$$

holds and

$$\varphi((\ell^i_{n+1}, Inv^i_{n+1}) \dots (\ell_{err}, Inv_{err})) \wedge I$$

is unsatisfiable for every $\pi_i \in \Pi_\ell$ then I is a path insensitive interpolant of ℓ in $\mathcal{P}$.

To enforce path insensitive interpolants we use a method to simplify the program graph without changing its correctness and without loosing 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 $((\ell, Inv), \varphi_1, (\ell_1, Inv_1)), \dots, ((\ell, Inv), \varphi_n, (\ell_n, Inv_n))$ 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. But LBE does not give us the desired full path insensitivity. It is rather a *partial path insensitivity.* Partial path insensitivity does not consider all execution paths through a location but a sub set. Especially for loops, it is intuitively clear that it is not possible to consider all execution paths through the loop header since we cannot encode all iteration of the loop. 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

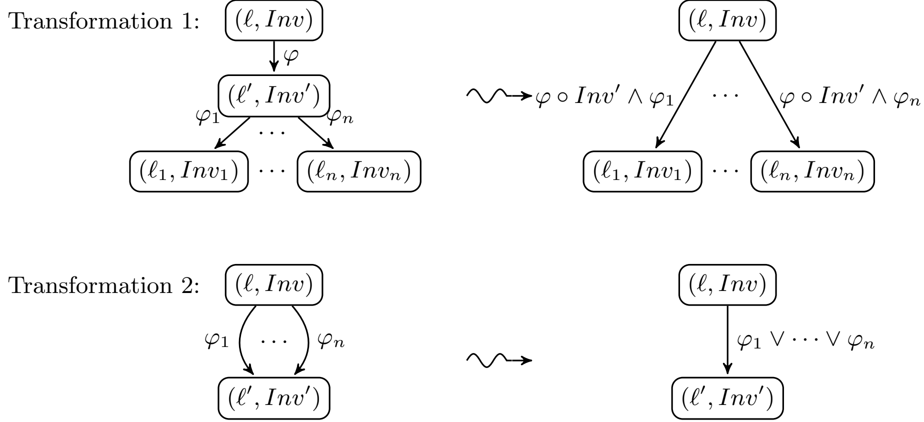


Fig. 7. The reduction rules for simplifying the program graph. Our implementation of large-block encoding follows closely [4]. Transformation 1 uses sequential composition to remove intermediate locations and Transformation 2 uses disjunction to remove multiple edges resulting from branches in the original program graph.

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 7. 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 program graph $\mathcal{P} = (Loc, \ell_{init}, \ell_{err}, \delta)$. For any location $\ell' \in Loc \backslash \{\ell_{init}, \ell_{err}\}$, s.t. the edge $((\ell, Inv), \varphi, (\ell', Inv'))$ exists uniquely, we reduce the program graph $\mathcal{P}$ as follows: 1) remove the location ℓ' from Loc, and 2) replace all pairs of edges $((\ell, Inv), \varphi, (\ell', Inv'))$, $((\ell', Inv'), \varphi', (\ell'', Inv''))$ by a new edge $((\ell, Inv), \varphi \circ Inv' \wedge \varphi', (\ell'', Inv''))$.*

Note that the rule duplicates the formulas φ and Inv' for every outgoing edge of location ℓ'. We can avoid exponential blow-up by replacing them by a boolean variable which are defined as φ resp. Inv', 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* $\mathcal{P} = (Loc, \ell_{init}, \ell_{err}, \delta)$ *and two nodes* $\ell, \ell' \in Loc$. *For any two edges* $((\ell, Inv), \varphi, (\ell', Inv'))$, $((\ell, Inv), \varphi', (\ell', Inv'))$ $\in \delta$, *we reduce the graph by replacing these edges by a new edge* $((\ell, Inv), \varphi \vee \varphi', (\ell', Inv'))$.

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

A proof for Theorem 3 is given in [4].

5 Experimental Evaluation

Our implementation is called ULTIMATE. It is based on the Eclipse RCP Framework. It allows to build tools as chains of plugins. For the purpose of experimental comparison we have also reimplemented the IMPACT [1] software model checker as a plugin in our framework. IMPACT is also an interpolation-based software model checker. But IMPACT builds an unwinding instead of manipulating the model itself. The framework allows us to compare the two approaches without changing the peripherals, i.e. input file, parser, SMT solver etc. In both cases we use Boogie PL [13] as input format and use SMTInterpol[1] as SMT solver. Subsequently, we denote our reimplementation of IMPACT as REIMPACT. As base for the comparison we use a set of simple examples (Table 1). The programs are all written in Boogie PL and have less than 100 lines of code. Their purpose is to emphasis the differences between our approach and the underlying approach of IMPACT. They also show the effect of path insensitive interpolation on both approaches. The tests were run on an AMD Athlon 64x2 Dual Core Processor with 2.50 Ghz and 4.00 GB of RAM. The operating system is an 64-bit Windows Server 2008 R2 Standard. Both approaches profit from the path insensitive interpolation. Without path insensitive interpolation, REIMPACT performs better than ULTIMATE since it returns results for more of the example programs. But with path insensitive interpolation, ULTIMATE returns more often a result than REIMPACT. Example 01 is an 4-bit counter with nested conditional branchings. And the final assertion states only about the highest bit. Therefore there are less

[1] `http://swt.informatik.uni-freiburg.de/research/tools/smtinterpol`

Table 1. The table shows the results of both model checking approaches on our example Boogie files. *Splits* shows how often nodes have been copied. *TPC* is number of theorem prover calls. Time is measured in millisec. *LBE* shows the results for path insensitive interpolation turned on(Yes) and off(No). Whereas, the symbols stand for ✔correct, ✗incorrect, – time out. The values in column *Splits*, *TPC* and *Time* relate to the runs with path insensitive interpolation turned on.

			Ultimate					reIMPACT				
No.	Nodes	Loops	Splits	TPC	Time	LBE Yes	LBE No	Splits	TPC	Time	LBE Yes	LBE No
01	14	1	0	3	34	✔	✔	3	1	35	✔	✔
02	5	1	3	33	288	✔	✔	4	8	108	✔	✔
03	7	2	13	156	2040	✔	–	–	–	–	–	–
04	36	1	42	766	13827	✔	–	6	52	1246	✔	–
05	31	1	1	16	462	✗	–	5	10	133	✗	–
06	11	1	6	85	9611	✔	–	–	–	–	–	–
07	26	1	0	3	66	✔	–	6	19	539	✔	–
08	50	1	0	3	79	✔	–	6	25	5447	✔	✔
19	98	1	0	3	116	✔	–	–	–	–	–	–
10	194	1	0	3	165	✔	–	–	–	–	–	–

paths to be checked and the assertion is a direct effect of the loop condition. As expected, both approaches perform similarly on this example. Example 02 has a single loop and no nested branching. The ART is a simple unrolling of the loop. But the splitting causes more nodes in the model and therefore also more theorem prover calls. Therefore, reIMPACT performs slightly better than Ultimate. Examples 04 is a standard planning example, gripper. The single loop can be considered as an event handler. Its nested conditional branchings are actions that can be performed. In each iteration only one of the branches can be taken. As in example 02, this reduces the number of paths. Due to the splitting, Ultimate produces more edges as the unwinding of reIMPACT does. Hence, we have more theorem prover calls. Example 05 is the same program as example 04 but with a bug insertion. reIMPACT is still faster but this time Ultimate shows comparable performance. Example 03 and 06(Example Sec. 2) have more complex branching structures. Additionally the necessary invariants are less obvious. It takes reIMPACT considerably more time to derive an invariant to proof the correctness of the programs. Examples 07 - 10 show the major advantage of our approach. The examples are 4-, 8-, 16-, 32-bit counters with a single loop that contain 4, 8, 16 and 32 `If-Then-Else` branchings, respectively. In contrast to examples 01-06, in examples 07-10 the control flow is not restricted by the semantics but depends on the current state of the variables. In these kind of programs our approach is much fast than reIMPACT. The manipulation of the model itself, as done by our splitting refinement, has the effect that obtained information of previous iterations is taken into account in the current iteration. This avoids the examination of paths that can't be taken by the branching any-

way. reIMPACT iterates through all permutations of the branching. In our approach the number of iterations remains the same and the additional time is only spend in the SMT solver. This also shows that the shifting of the branching into the formula is handled by the solver very well. Overall, the examples show that the combination of splitting with path insensitive interpolation has potential. On our experimental set of examples it terminates more often than the IMPACT approach. The slower performance on some examples, results from creation of unnecessary edges in the splitting. This can be optimized further.

6 Related Work

We compare our tool to modern software model checking tools like Blast [5], CPAchecker [6], and IMPACT [16]. The most common approaches, e.g. Blast, SLAM [2], SATABS [10], use CEGAR algorithms with predicate abstraction. In predicate abstraction the model is refined by newly obtained predicates e.g. derived from interplants. In our approach we use the interpolants themselves in order to refine the model. This technique has been introduced by Ken McMillan [14]. In [16] 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. Wolverine [12]. The main difference to our approach is the abstraction technique. In contrast to our approach, IMPACT uses an abstract reachability tree. Our approach modifies the model itself. In this model the gathered information is reused on every path examination. This approach is based on slicing abstraction [8]. Slicing abstraction was implemented in a tool called SLAB [11]. 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.

7 Conclusion

We have introduced the concept of path insensitive interpolation. We also presented a first approach to use path insensitive interpolation in combination with splitting refinement to derive loop invariants. We demonstrated with an experimental implementation that it is in fact efficient to burden the SMT solver the task to find useful interpolants by providing it more information about the program. We did this by using large block encoding as a compression technique and computing path interpolants on the compressed model. We showed that, although just partially path insensitive, we obtain promising results with this approach.

Applying LBE on the entire model allows us a partial path insensitivity. But the future work will be to find techniques to improve the path insensitivity. This can be done by computing interpolants not over the paths of the compressed model but by focusing on single locations. If considering a single location, one can summarize the entire prefixes of the paths leading from the initial location to the observed location and also summarize all suffixes leading from the oberserved location to the error location. But even this approach would not be fully path insensitive since we can still not take all loop iteration into consideration. Additionally this appraoch might cause a lot of redundant computations in the SMT solver. Instead of one SMT solver call per path, we would cause one per each location. This would get too costly. For this purpose, the communication between the SMT solver and the software model checker must be improved. A steering by the software model checker could also effect the quality of the interpolants. Further, an optimization of the splitting refinement would reduce the number of theorem prover calls and increase the performance of the approach.

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] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.

[7] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In E. Emerson and K. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer Berlin / Heidelberg, 2006.

[8] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. In F. Arbab and M. Sirjani, editors, *FSEN*, 2007.

[9] R. Bruttomesso, S. Ghilardi, and S. Ranise. Rewriting-based quantifier-free interpolation for a theory of arrays. In *RTA*, pages 171–186, 2011.

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

[11] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS*, LNCS. Springer-Verlag, 2010.

[12] D. Kroening and G. Weissenbacher. Interpolation-based software verification with wolverine. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 573–578. Springer Berlin / Heidelberg, 2011.

[13] K. R. M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at `http://research.microsoft.com/~leino/papers.html`.
[14] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pages 1–13, 2003.
[15] K. L. McMillan. Applications of Craig interpolants in model checking. In *TACAS*, pages 1–12, 2005.
[16] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.