Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems*

Javier Esparza esparza@in.tum.de
Stefan Kiefer kiefer@in.tum.de
Stefan Schwoon schwoon@in.tum.de
Institut für Informatik,
Technische Universität München,
Germany

Abstract

Counterexample-guided abstraction refinement (CEGAR) has proven to be a powerful method for software model checking. In this paper, we investigate this concept in the context of sequential (possibly recursive) programs whose statements are given as Binary Decision Diagrams (BDDs). We examine how Craig interpolants can be computed efficiently in this case and propose a new special type of interpolants. Moreover, we show how to treat multiple counterexamples in one refinement cycle. We have implemented this approach within the model checker Moped and report on experiments.

Keywords: *model-checking, abstraction refinement, Craig interpolation, BDDs, pushdown systems*

Submitted January 2007; revised March 2008; published June 2008

1. Introduction

CEGAR is a powerful tool for automated abstraction of hardware and software systems. Originally designed for the verification of hardware designs, this technique has been successfully utilized for software verification as well. Particularly, the SLAM project [1] has gained attention and has demonstrated the effectiveness of software verification for device drivers. The BLAST tool [8] and the MAGIC tool [2] have been applied successfully in the domains of security protocols and real-time operating-system kernels.

The CEGAR paradigm was introduced in [3]. The goal is to check if a given concrete program can reach a certain *error label*. Since the data space of the concrete program is too large, its size is reduced with a predicate abstraction method. Initially, there are no predicates, therefore the initial abstraction is very coarse (no data, only control flow). This abstract program is then model checked.

Since the abstract program is, by construction, an overapproximation of the concrete one, model checking it can have two possible outcomes: In the first case the error label is not reachable; then we know that it is not reachable in the concrete program either and the

* This work was partially supported by the DFG project *Algorithms for Software Model Checking*. Parts of this work appeared in a TACAS'06 paper [5] and a Master's thesis [11]. Most of this work was done at the Universität Stuttgart.

CEGAR process terminates. In the second case the error label is reachable in the abstract program, as illustrated by a counterexample, i.e., a path leading to the error label. Due to the overapproximation, this path may be *spurious*, i.e., not realizable in the concrete system. If it is spurious, then suitable new predicates need to be introduced to refine the abstraction such that this counterexample is excluded in future predicate abstractions. If it is not spurious (*real* counterexample), then it can be reported to the user, and the process terminates.

This process continues in cycles, until the abstraction is fine enough to either conclude that the error label is unreachable or that a real counterexample exists.

1.1 Our Work and Related Work

We develop a CEGAR scheme for *symbolic pushdown systems* (SPDS), i.e. systems where the control flow is described by a pushdown system and where BDDs describe how variables are changed by the program statements. We integrate our approach into Moped [15], a model checker that works with the SPDS model.

From a high-level perspective, our approach can be characterized as follows: We first translate a program with integer variables to a program with finitely many variable bits (e.g. 8 or 16 bits per variable). A similar translation is done by compilers for most programming languages, including C and Java, where the language specification requires a fixed-bit representation of integer variables. If we use fewer bits than the programming language specification, we might lose some bugs that occur only with large numbers.[1] Then we use CEGAR to reduce the state space. No precision is lost by CEGAR, because the abstraction is appropriately refined during the process. This is the main advantage of the CEGAR approach over the use of shorter bit vectors. Notice that recursion may induce an infinite state space.

The input for our CEGAR scheme is essentially a sequential program with procedures (potentially recursive) whose variables are represented by a finite number of bits. BDDs capture the modification of the variables through the program statements. The problem is whether this program can reach a specific error label or not.

Moped could be directly used for this problem, but we use a CEGAR scheme to reduce its resource consumption. Our abstract programs are Boolean programs whose variables are previously introduced predicates. The statements of the abstract programs modify the truth values of the predicates. This is again captured by BDDs. Those abstract programs are checked using Moped.

The consequent use of BDDs throughout the CEGAR process distinguishes our work from related work about CEGAR in software. For instance, in the SLAM project [1], a BDD-based model checker is employed on the abstract level, but symbolic expression representations together with theorem provers are applied on the concrete level. MAGIC [2] does not use BDDs at all, but relies on SAT solvers and theorem provers. Also [8, 7] make use of theorem provers, whereas we use BDD technology for the concrete program, the abstract programs, and for the predicates in our abstraction mechanism. We therefore avoid theorem provers, which assume infinite ranges of integer variables and often form bottlenecks in related projects, e.g. in [1]. Another argument for the use of BDDs as

[1] Our current implementation does not model overflows etc. faithfully, so some overflow bugs may be lost.

opposed to theorem provers is the fact that BDDs could be directly used for modeling computer arithmetic modulo 2^{32}, although, unfortunately, this is not yet supported by our current implementation. Using bit vectors instead of infinite-range integers has also been successfully used for the verification of hardware, see e.g. [9].

Another feature of our work is the use of multiple counterexamples in a single refinement step. Moped constructs a "witness graph" (see [14]) which, in the model checking phase, records information about which program states can be reached via which previously reached program states. When viewed from the perspective of the error label, this graph is a DAG containing possible (abstract) error traces. We use not only a single counterexample for abstraction refinement, but this DAG. If the counterexample DAG contains a real (non-spurious) counterexample, it is reported. Otherwise we compute predicates ensuring that none of the counterexamples in the DAG will occur again in future abstractions. In [6], multiple counterexamples are also used in a CEGAR scheme, but not for software and not in a DAG structure. A particular example illustrating the usefulness of the DAG approach is given in Section 6.4.

For the predicate generation, we use Craig interpolation (see [7, 13]). In contrast to [7], we consider Craig interpolation for pure propositional logics. We show that the computation of Craig interpolants works well with BDDs and that their use gives us the flexibility for choosing heuristics about *which* interpolants to use, since Craig interpolants are, in general, not unique. In particular, our heuristics for choosing interpolants do not depend on the internal strategies of a SAT solver or a theorem prover.

Organization of the paper. This paper proceeds as follows. In Section 2 we investigate Craig interpolation for propositional logics and derive computation schemes that are suitable for BDDs. In Section 3, symbolic pushdown systems, a model for sequential programs, are reviewed. In Section 4, the techniques of Section 2 are applied to the computation of predicates that rule out DAGs of abstract counterexamples. Section 5 sketches our predicate abstraction scheme. We give evidence for the usefulness of our concepts in Section 6 and conclude in Section 7.

2. Craig Interpolation

In [12, 7], Craig interpolation was used to automatize abstraction refinement. As in [12] (and in contrast to [7], where a specialized arithmetic proof system is used) we are interested in Craig interpolants for pure propositional logic. We write $Occ(F)$ for the set of variables that occur (syntactically) in a formula F.

Definition 2.1 *Let (F, G) be a pair of formulas with $F \wedge G$ unsatisfiable. A (syntactic) interpolant for (F, G) is a formula I s.t. F implies I (written: $F \models I$), $I \wedge G$ is unsatisfiable and $Occ(I) \subseteq Occ(F) \cap Occ(G)$.*

Example 2.2 *The formula $I = x$ is an interpolant for $(x \wedge y, \neg x \wedge z)$.*

Craig's Interpolation Theorem [4] states that interpolants always exist, but they are not unique. In [12], interpolants are obtained from a resolution proof of the unsatisfiability of $F \wedge G$, which is, in turn, constructed by a SAT solver. However, in our BDD-based setting this result is no longer useful, because we do not prove unsatisfiability of $F \wedge G$ by means of a SAT solver. We show that there exist interpolants that do not depend on the internal strategies of a SAT solver or a theorem prover, and can be naturally computed by standard BDD operations.

2.1 Strongest and Weakest Interpolants

It is easy to see that if I and I' are interpolants for (F, G), then so are $I \wedge I'$ and $I \vee I'$. It follows that "the strongest interpolant" and "the weakest interpolant", as defined below, exist and are unique.

Definition 2.3 *Let $F \wedge G$ be unsatisfiable. The strongest interpolant for (F, G), denoted $SI(F, G)$, is the unique interpolant for (F, G) that implies any other interpolant. The weakest interpolant for (F, G), denoted $WI(F, G)$, is the unique interpolant that is implied by any other interpolant.*

Clearly, $SI(F, G) \models WI(F, G)$ holds. Proposition 2.4 below shows that $SI(F, G)$ and $WI(F, G)$ can be obtained by standard BDD operations (quantification over variables). If F and G are formulas, we define the notation $F \uparrow G := \exists(Occ(F) \setminus Occ(G)).F$ and $F \downarrow G := \forall(Occ(F) \setminus Occ(G)).F$. Notice that $F \downarrow G \models F \models F \uparrow G$ always holds. Intuitively, the formula $F \uparrow G$ is a weakened version of F, such that it contains only variables that also occur in G. Similarly, the formula $F \downarrow G$ is a strengthened version of F, such that it contains only variables that also occur in G.

Proposition 2.4 (Strongest and Weakest Interpolants) *Let $F \wedge G$ be unsatisfiable. Then $SI(F, G) \equiv F \uparrow G$ and $WI(F, G) \equiv (\neg G) \downarrow F$.*

The proof uses the following lemma.

Lemma 1 *$F \uparrow G$ is the strongest formula among those formulas that are implied by F and contain only variables in $Occ(G)$. Analogously, $(\neg G) \downarrow F$ is the weakest formula among those formulas that imply $\neg G$ and contain only variables in $Occ(F)$.*

Proof of the lemma. Let $I = F \uparrow G$. Let J be any formula that is implied by F and contains only variables in G. We show $I \models J$: Let $\mathcal{A}$ be any assignment s.t. $\mathcal{A}(I) = \textbf{true}$. Then, $\mathcal{A}$ can be extended to an assignment $\mathcal{A}^+$ s.t. $\mathcal{A}^+(F) = \textbf{true}$. Since $F \models J$, we have $\mathcal{A}^+(J) = \textbf{true}$ and $\mathcal{A}(J) = \textbf{true}$.

The statement about $(\neg G) \downarrow F$ is proved analogously. □

Proof of Proposition 2.4. Let $I = F \uparrow G$. As $SI(F, G)$ is, by definition, implied by F and contains only variables in $Occ(G)$, we have $I \models SI(F, G)$ by the lemma. But, as $F \models I \models SI(F, G) \models \neg G$, the formula I is an interpolant for (F, G). Hence, $SI(F, G) \models I$, so $SI(F, G) \equiv I$.

Analogously, one shows $WI(F, G) \equiv (\neg G) \downarrow F$. □

Example 2.5

$$\begin{aligned}
SI(w \wedge x \wedge y, \neg x \wedge \neg y \wedge \neg z) &\equiv (w \wedge x \wedge y) \uparrow (\neg x \wedge \neg y \wedge \neg z) \\
&\equiv \exists w.(w \wedge x \wedge y) \\
&\equiv x \wedge y \\
WI(w \wedge x \wedge y, \neg x \wedge \neg y \wedge \neg z) &\equiv (x \vee y \vee z) \downarrow (w \wedge x \wedge y) \\
&\equiv \forall z.(x \vee y \vee z) \\
&\equiv x \vee y
\end{aligned}$$

2.2 Iterative Computation of Interpolants

In this section, we consider the following problem: Given a formula $F = F_1 \wedge \ldots \wedge F_n$, determine whether F is unsatisfiable, and if so, find interpolants for the pairs $(F^{..i}, F^{i+1..})$, $i \in \{1, \ldots, n\}$, where $F^{..i} := F_1 \wedge \ldots \wedge F_i$ and $F^{i+1..} := F_{i+1} \wedge \ldots \wedge F_n$.

We sketch the motivation here, cf. [7]. Each formula F_i models a program instruction. A formula $F = F_1 \wedge \ldots \wedge F_n$ models a trace through a program. In order to check if the trace is feasible or spurious, one can check if F is satisfiable or unsatisfiable. If the trace is spurious (unsatisfiable F), then we would like to find an explanation for the spuriousness. Suitable interpolants give such an explanation and are used for a refined predicate abstraction that no longer allows for the spurious trace. More details are given in Section 4, where we will apply the results of this section.

In the following we show that strongest and weakest interpolants for $(F^{..i}, F^{i+1..})$ can be computed iteratively. In the program model, strongest and weakest interpolants correspond to strongest postconditions (of **true**) and weakest preconditions (of **false**).

Proposition 2.6 *Let $F = F_1 \wedge F_2 \wedge \ldots \wedge F_n$ be a formula and let $F^{..i}$ and $F^{i+1..}$ be defined as above. Let $\{I_i\}$ and $\{J_i\}$ be families of predicates defined according to the following procedures:*
$I_0 := \mathbf{true}, I_{i+1} := (I_i \wedge F_{i+1}) \uparrow F^{i+2..}$ *for* $i = 0, \ldots, n-1$, *and*
$J_n := \mathbf{false}, J_{i-1} := (F_i \rightarrow J_i) \downarrow F^{..i-1}$ *for* $i = n, \ldots, 1$.

*(i) F is unsatisfiable iff $I_n \equiv$ **false** iff $J_0 \equiv$ **true**.*

(ii) If F is unsatisfiable, then $I_i \equiv SI(F^{..i}, F^{i+1..})$ and $J_i \equiv WI(F^{..i}, F^{i+1..})$.

Proof. We first show by induction that $I_i \equiv F^{..i} \uparrow F^{i+1..}$. In the base case we have $I_0 \equiv \mathbf{true} \equiv \mathbf{true} \uparrow F^{1..}$. For the induction step, let $i + 1 > 0$. Then we have

$$\begin{aligned}
I_{i+1} &\equiv (I_i \wedge F_{i+1}) \uparrow F^{i+2..} && \text{(definition } I_{i+1}) \\
&\equiv ((F^{..i} \uparrow F^{i+1..}) \wedge F_{i+1}) \uparrow F^{i+2..} && \text{(induction hypothesis)} \\
&\equiv ((F^{..i} \wedge F_{i+1}) \uparrow F^{i+1..}) \uparrow F^{i+2..} && \text{(the variables not occurring in } F^{i+1..} \\
& && \quad \text{do not occur in } F_{i+1}) \\
&\equiv (F^{..i} \wedge F_{i+1}) \uparrow F^{i+2..} && (\mathit{Occ}(F^{i+2..}) \subseteq \mathit{Occ}(F^{i+1..})) \\
&\equiv F^{..i+1} \uparrow F^{i+2..} && \text{(definition } F^{..i+1}) \; .
\end{aligned}$$

Now, (i) follows immediately and (ii) is a consequence of Proposition 2.4. The statements about J_i are proved analogously. □

Given $F = F_1 \wedge \ldots \wedge F_n$, we can iteratively compute BDDs for the sequence I_i or J_i with the above procedure. We can decide if F is satisfiable using (i). If F is unsatisfiable, then, by (ii), we have computed $SI(F^{..i}, F^{i+1..})$ or $WI(F^{..i}, F^{i+1..})$.

For our CEGAR purposes, we will need the following property:

Definition 2.7 (Tracking Property) *Let $F_1 \wedge \ldots \wedge F_n$ be unsatisfiable, and let K_i be interpolants for $(F^{..i}, F^{i+1..})$. We say that the family $\{K_i\}$ satisfies the* tracking property *if $K_i \wedge F_{i+1} \models K_{i+1}$.*

Proposition 2.8 *Let $F_1 \wedge F_2 \wedge \ldots \wedge F_n$ be unsatisfiable. Let $\{I_i\}$ and $\{J_i\}$ be families of predicates defined according to the following procedures:*
*$I_0 :=$ **true**, $I_{i+1} :=$ any interpolant for $(I_i \wedge F_{i+1}, F^{i+2..})$, where $i = 0, \ldots, n-1$,*
*$J_n :=$ **false**, $J_{i-1} :=$ any interpolant for $(F^{..i-1}, \neg(F_i \rightarrow J_i))$, where $i = n, \ldots, 1$.*
Then $\{I_i\}$ and $\{J_i\}$ are interpolants for $(F^{..i}, F^{i+1..})$ and satisfy the tracking property.

Proof. We prove the statement about $\{I_i\}$ by induction over i. In the base case we have $I_0 \equiv$ **true**, which is an interpolant for (**true**, $F^{1..}$), because $F^{1..}$ is unsatisfiable. For the induction step, let $i + 1 > 0$. By the induction hypothesis, I_i is an interpolant for $(F^{..i}, F^{i+1..})$, so $I_i \wedge F_{i+1} \wedge F^{i+2..}$ is unsatisfiable. Therefore, I_{i+1} is well-defined. By definition of I_{i+1}, the tracking property ($I_i \wedge F_{i+1} \models I_{i+1}$) holds. Furthermore, we have

$$\begin{aligned} F^{..i+1} &\equiv F^{..i} \wedge F_{i+1} & & \\ &\models I_i \wedge F_{i+1} & & \text{(by induction hypothesis: } F^{..i} \models I_i) \\ &\models I_{i+1} & & \text{(by definition of } I_{i+1}) \\ &\models \neg F^{i+2..} & & \text{(by definition of } I_{i+1})\,. \end{aligned}$$

Hence, I_{i+1} is an interpolant for $(F^{..i+1}, F^{i+2..})$. The statement about $\{J_i\}$ is proved analogously. □

Corollary 2.9 *$\{SI(F^{..i}, F^{i+1..})\}$ and $\{WI(F^{..i}, F^{i+1..})\}$ satisfy the tracking property.*

Finally, Proposition 2.10 shows the interplay between interpolants and disjunction:

Proposition 2.10

(i) If $(F \vee G) \wedge H$ is unsatisfiable, then $SI(F \vee G, H) \equiv SI(F, H) \vee SI(G, H)$.

(ii) If $F \wedge (G \vee H)$ is unsatisfiable, then $WI(F, G \vee H) \equiv WI(F, G) \wedge WI(F, H)$.

Proof.

(i) Let Z be the variables not occurring in H.
Then $SI(F \vee G, H) \equiv \exists Z.(F \vee G) \equiv \exists Z.F \vee \exists Z.G \equiv SI(F, H) \vee SI(G, H)$.

(ii) Analogously. □

2.3 Conciliated Interpolants

Interpolants can be seen as explanations indicating why counterexamples are spurious. It makes sense to look for "simple" explanations. It seems reasonable to consider an interpolant "simple" if few variables occur in it. Since we work with BDD libraries, it is natural to strengthen the notion of occurrence semantically:

Definition 2.11 *A variable v* occurs semantically *in F if $\exists v.F \not\equiv F$. The set of variables that occur semantically in F is denoted by $Supp(F)$.*

One could strengthen the notion of interpolants accordingly (by replacing *Occ* by *Supp* in Def. 2.1). Such *semantic* interpolants are also *syntactic* interpolants. We now show that one can find simpler interpolants than the weakest and strongest ones, still using only quantifications. If I and J are strongest and weakest (syntactic or semantic) interpolants for (F, G), respectively, then we have $F \models I \models J \models \neg G$, but not necessarily $\mathit{Supp}(I) = \mathit{Supp}(J)$. Now we can compute the strongest and weakest *semantic* interpolants I_1, J_1 for the pair $(I, \neg J)$. Since $F \models I \models I_1 \models J_1 \models J \models \neg G$, we have that I_1 and J_1 are also interpolants for (F, G). If $\mathit{Supp}(I) \neq \mathit{Supp}(J)$, then at least one of I_1 and J_1 will be simpler than I and J, since the variables in the symmetric difference are quantified out. This simplification procedure can be iterated until a pair I_n, J_n is reached such that $\mathit{Supp}(I_n) = \mathit{Supp}(J_n)$.

Definition 2.12 *Let (F, G) be formulas over a set V of variables s.t. $F \wedge G$ is unsatisfiable, and let $Z \subseteq V$ s.t. $\exists Z.F$ and $\forall Z.\neg G$ are interpolants for (F, G). We say that $\exists Z.F, \forall Z.\neg G$ are* conciliated interpolants *if $\mathit{Supp}(\exists Z.F) = \mathit{Supp}(\forall Z.\neg G)$. We call $\mathit{Supp}(\exists Z.F)$ a* conciliating set *in this case.*

Example 2.13 *Strongest and weakest interpolants are not necessarily conciliated:*
For $F = x \wedge (y \vee z)$ and $G = \neg(x \vee y \vee (z \wedge w))$, we have $\mathit{SI}(F, G) \equiv F$ and $\mathit{WI}(F, G) \equiv x \vee y$, which are not conciliated. The formula x is a conciliated interpolant for (F, G), and $\{x\}$ is a conciliating set.

The algorithm in Figure 1 computes a pair of conciliated interpolants.

```
function conciliate(formulas $F, G$) returns $(Z, \exists(V \setminus Z).F, \forall(V \setminus Z).\neg G)$
/* $F \wedge G$ unsatisfiable is an input requirement */
/* $Z$ is the maximal conciliating set */
$I := F$;   $J := \neg G$;   $Z := \mathit{Supp}(F) \cup \mathit{Supp}(G)$
repeat  $X := \mathit{Supp}(I) \setminus \mathit{Supp}(J)$;   $I := \exists X.I$;   $Z := Z \setminus X$
        $Y := \mathit{Supp}(J) \setminus \mathit{Supp}(I)$;   $J := \forall Y.J$;   $Z := Z \setminus Y$
until $Y = \emptyset$
return $(Z, I, J)$
```

Figure 1. Computation of conciliated interpolants for the maximal conciliating set

Given a pair of formulas, the pair of conciliated interpolants is not unique. Proposition 2.14 characterizes the pair computed by the algorithm.

Proposition 2.14

(i) *Let (I_1, J_1) and (I_2, J_2) be two pairs of conciliated interpolants, and let C_1, C_2 be the corresponding conciliating sets. Then $C_1 = C_2$ if and only if $I_1 \equiv I_2$ and $J_1 \equiv J_2$.*

(ii) *Conciliating sets are closed under union, but not under intersection.*

(iii) *There is a unique maximal conciliating set.*

(iv) *The algorithm of Figure 1 computes the unique maximal conciliating set.*

Proof.

(i) "$\Leftarrow$": Clearly, by Def. 2.12, a pair of conciliated interpolants (I, J) determines their unique conciliating set $\mathit{Supp}(I) = \mathit{Supp}(J)$.
"$\Rightarrow$": Let $V = \mathit{Supp}(F) \cup \mathit{Supp}(G)$. Let $C \subseteq V$ be a conciliating set for (F, G), i.e., let $C = \mathit{Supp}(I) = \mathit{Supp}(J)$ with $I = \exists Z.F$ and $J = \forall Z.\neg G$ for some set $Z \subseteq V$. We show $I \equiv \exists(V \setminus C).F$. (Showing $J \equiv \forall(V \setminus C).\neg G$ is analogous.)
By definition of I, the sets Z and C must be disjoint, i.e., $Z \subseteq V \setminus C$. By definition of C, no variable in $V \setminus C$ occurs (semantically) in I. Combining these facts yields $I \equiv \exists(V \setminus C).I \equiv \exists((V \setminus C) \cup Z).F \equiv \exists(V \setminus C).F$.

(ii) We first show the closure under union. Let $F \models \neg G$ and $V \equiv \mathit{Supp}(F) \cup \mathit{Supp}(G)$. Let X and Y be conciliating sets and $Z = X \cup Y$. With X conciliating and (i), we have

$$F \models \exists(V \setminus Z).F \models \exists(V \setminus X).F \models \forall(V \setminus X).\neg G \models \forall(V \setminus Z).\neg G \models \neg G\,.$$

So, $\exists(V \setminus Z).F$ and $\forall(V \setminus Z).\neg G$ are interpolants for (F, G). Thus, for Z to be conciliating, it remains to show:

$$\mathit{Supp}(\exists(V \setminus Z).F) = \mathit{Supp}(\forall(V \setminus Z).\neg G).$$

Let $v \in V$.

Case 1: $v \in Z$. Then w.l.o.g. $v \in X$. Since X is conciliating, we have $v \in \mathit{Supp}(\exists(V \setminus X).F) = \mathit{Supp}(\forall(V \setminus X).\neg G)$,
thus $v \in \mathit{Supp}(\exists(V \setminus Z).F)$ and $v \in \mathit{Supp}(\forall(V \setminus Z).\neg G)$.

Case 2: $v \notin Z$.
Then $v \notin \mathit{Supp}(\exists(V \setminus Z).F)$ and $v \notin \mathit{Supp}(\forall(V \setminus Z).\neg G)$.

Hence, in both cases we have

$$v \in \mathit{Supp}(\exists(V \setminus Z).F) \Longleftrightarrow v \in \mathit{Supp}(\forall(V \setminus Z).\neg G).$$

Thus, Z is conciliating.

Conciliating sets are not closed under intersection. Consider $F \equiv (x \wedge y) \vee z$ and $\neg G \equiv x \vee y \vee z$. The sets $\{x, z\}$ and $\{y, z\}$ are conciliating, but $\{z\}$ is not conciliating, because $\exists\{x, y\}.F \equiv \textbf{true}$ and $\forall\{x, y\}.\neg G \equiv z$.

(iii) Follows directly from (ii).

(iv) The algorithm removes, in each iteration, only variables from Z that clearly cannot occur in any conciliating set. In addition, whenever a variable is quantified out s.t. it no longer occurs in I or J, then it is also removed from Z. It follows that upon termination of the algorithm, Z is the maximal conciliating set. Because we have finitely many variables, the algorithm indeed terminates. □

One may argue that, since we are interested in simple interpolants, we would like to compute a *minimal* conciliating set. Unfortunately, in general there is no unique minimal set, as can be seen by the example at the end of above proof, part (ii). We can still compute conciliated interpolants for a minimal conciliating set by a greedy algorithm as shown in Figure 2. Even though the resulting conciliating set is minimal in the sense that no proper subset is conciliating, we do not claim that we find the "best" conciliated interpolants by that algorithm: The algorithm is nondeterministic in the selection of the variables z, and we do not have a strategy that selects "the best" next variable to be quantified out.

```
function conciliateMinimal(formulas $F, G$) returns $(Z, \exists(V \setminus Z).F, \forall(V \setminus Z).\neg G)$
/* $F \wedge G$ unsatisfiable is an input requirement */
/* $Z$ is a minimal conciliating set */
$I := F$;   $J := \neg G$
while exists $z \in Z$ such that $\exists\{z\}.I \models \forall\{z\}.J$ do
    $(Z, I, J) :=$ conciliate($\exists\{z\}.I, \neg\forall\{z\}.J$)
end while
return $(Z, I, J)$
```

Figure 2. Computation of conciliated interpolants for a minimal conciliating set

In the context of abstraction refinement, we use the algorithm from Figure 1 as interpolation (and simplification) method when computing a family of interpolants according to Proposition 2.8. Recall that this proposition guarantees the tracking property, regardless of the interpolation procedure.

3. Symbolic Pushdown Systems

As our program model, we use symbolic pushdown systems (SPDSs) [15].

Definition 3.1 (SPDS) [2] *An SPDS is a quadruple* $(G, \Gamma \times L, \Delta, \gamma_0)$*, where*

- $G = \{\mathbf{true}, \mathbf{false}\}^{n_G}, n_G \geq 0$*, is the set of global variable valuations,*
- Γ *is a set of* control points*,*
- $L = \{\mathbf{true}, \mathbf{false}\}^{n_L}, n_L \geq 0$*, is the set of local variable valuations,*
- Δ *is a set of* symbolic transition rules*, where each rule is of the form* $\langle\gamma\rangle \hookrightarrow \langle\gamma_1, \ldots, \gamma_n\rangle$ (R) *with* $0 \leq n \leq 2, \gamma, \gamma_1, \ldots, \gamma_n \in \Gamma_0$ *and* $R \subseteq (G \times L) \times (G \times L^n)$*,*
- $\gamma_0 \in \Gamma$ *is the start address.*

SPDSs model programs with (possibly recursive) procedures and with finite data types. Γ corresponds to the set of control points in a program, while G and L represent the possible values of global and local variables, respectively. A *configuration* of an SPDS is a tuple

[2] This definition is slightly more restrictive than in [15], because it does not include explicit control states. However, no expressive power is lost, because control states could be encoded in the relations R.

$\langle g, (\gamma_1, \ell_1) \cdots (\gamma_n, \ell_n) \rangle \in G \times (\Gamma \times L)^*$, where γ_1 is the current program counter, g is the current global store, and ℓ_1 are the current local variables. Thus, an SPDS can faithfully model the control-flow of a program even in the presence of recursion, provided that the data (global and local) accessible at any one point during execution is finite. Such an SPDS can be obtained, for instance, by translation from Java bytecode using jMoped [17, 16].

The rules model statements in a programming language. The relation R of a rule describes the relation between the variables before and after execution of the rule.[3] In our setting, they are given as BDDs. The right-hand side of the rules can consist of zero, one or two control points. Whereas a rule with one control point on the right-hand side describes an intraprocedural statement, a rule with two control points on the right-hand side describes a procedure call, a *push*: γ_1 is the start address of the callee and γ_2 the return address of the caller. Parameter passing can be encoded in the relation R by initializing the local variables of the callee. A rule with zero control points on the right-hand side is the termination of a procedure, a *pop*. Return values can be encoded in the relation R by restricting the global variables. SPDSs are also discussed in greater detail in [15].

Example 3.2 *Consider the procedures in Figure 3. The procedure* `m` *calls the procedure* `f`*. Procedure* `f` *returns a value using the global variable G. Procedure* `m` *has a local variable L, procedure* `f` *has a local variable A. The transition rules of a corresponding SPDS are shown on the right-hand side. The start address is $m0$. Non-primed variables refer to the variable value* before *execution of the rule. Primed and double-primed variables refer to the variable value* after *execution of the rule. Double-primed variables only occur at push rules and belong to variables of the second control point on the right-hand side of a push rule (in the example: L'' belongs to $m2$).*

```
procedure m
m0:  $L := L \cdot (L+1)$
m1:  call f($L$)
m2:  if $G \neq 0$ then goto error
```

$$\begin{array}{lll} \langle m0 \rangle \hookrightarrow \langle m1 \rangle & (L' = L \cdot (L+1) \ \wedge \ G' = G) \\ \langle m1 \rangle \hookrightarrow \langle f0, m2 \rangle & (L'' = A' = L \ \wedge \ G' = G) \\ \langle m2 \rangle \hookrightarrow \langle error \rangle & (L' = L \wedge G \neq 0 \ \wedge \ G' = G) \end{array}$$

```
procedure f($A$)
f0:  if $A$ even then
f1:        $A := 0$
f2:  else $A := 561$
f3:  $G := A$
```

$$\begin{array}{ll} \langle f0 \rangle \hookrightarrow \langle f1 \rangle & (A \text{ even} \ \wedge \ A' = A \ \wedge \ G' = G) \\ \langle f1 \rangle \hookrightarrow \langle f3 \rangle & (A' = 0 \ \wedge \ G' = G) \\ \langle f0 \rangle \hookrightarrow \langle f2 \rangle & (A \text{ odd} \ \wedge \ A' = A \ \wedge \ G' = G) \\ \langle f2 \rangle \hookrightarrow \langle f3 \rangle & (A' = 561 \ \wedge \ G' = G) \\ \langle f3 \rangle \hookrightarrow \langle \rangle & (G' = A) \end{array}$$

Figure 3. Two simple procedures along with an equivalent SPDS

Moped can model check such a concrete SPDS. However, in our CEGAR scheme we use Moped only to model check Boolean SPDSs that have the same control flow structure, but overapproximate the given concrete SPDS.

[3] To avoid deadlocks in an SPDS, one should make sure that for all (g, l) there is a $((g, l), (g', l^1, \ldots, l^n)) \in R$. This is guaranteed when real programs are translated into the SPDS model.

4. Computing Predicates for a DAG of Counterexamples

We use Moped to model check the (abstract) SPDSs generated in our refinement cycle. If Moped finds that the error label is reachable in a given SPDS, it constructs a DAG that illustrates the abstract paths leading to the error (see [14] for details on this construction). In brief, the nodes of the DAG are the configurations of the SPDS, the arcs are labeled by symbolic transition rules. There is a single "sink" node with no outgoing arcs, the error configuration.

For instance, consider the program in Figure 4. In the initial abstraction, all data is discarded, therefore Moped finds two counterexamples, one that does not enter the loop body, and one that enters it exactly once. The resulting counterexample DAG produced by Moped is shown on the right-hand side of Figure 4. (For the time being, ignore the predicates in curly brackets.)

Once we have the DAG, we discard the information about the abstract variable values and replace the abstract rules by their concrete counterparts. We then need to decide if all counterexamples in the DAG are spurious or if there exists a real one. We call the DAG spurious in the first case. For instance, the DAG in Figure 4 is spurious.

```
1:  $X := X \cdot (X+1)$
2:  while $Y$ odd do
3:      $Y := Y + 1$
4:  if $(X+Y)$ odd
        then goto error
5:  end
```

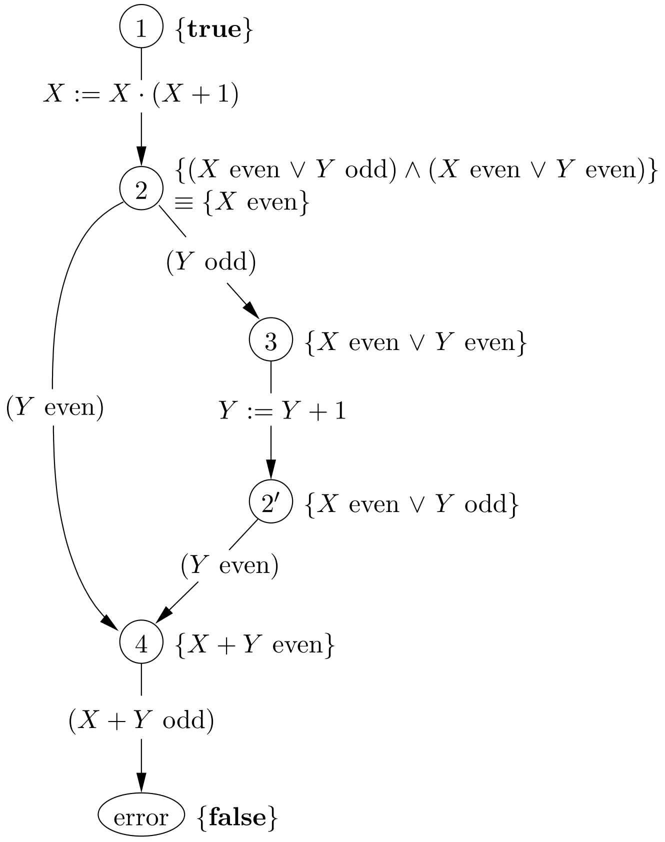


Figure 4. Program and counterexample DAG with weakest interpolants

Let D be a DAG for the rest of the section. We describe our predicate generation method in three steps: (i) for single counterexamples without procedures, (ii) for counterexample

DAGs without procedures, and (iii) for counterexample DAGs with a procedural structure. In all cases, we proceed as follows:

- We construct a so-called *characteristic formula* F_D that is unsatisfiable if and only if the DAG is spurious.
- For each node n in D, we compute a predicate P_n such that the following holds. For every arc in D, say the arc is from n_1 to n_2 and is labeled by a rule with relation R, the triple $\{P_{n_1}\}\ R\ \{P_{n_2}\}$ is a valid Hoare triple (recall that a SPDS rule R corresponds to a program instruction). For example, in Figure 4, the triple

  $$\{X \text{ even } \vee\ Y \text{ even}\}\ Y := Y + 1\ \{X \text{ even } \vee\ Y \text{ odd}\}$$

  is a valid Hoare triple.
- We show that unsatisfiability of F_D can be decided by computing and examining these predicates P_n. If F_D is unsatisfiable, i.e., if D is spurious, then the predicates explain the infeasibility of the traces of D, and adding them in future abstractions excludes those traces.

4.1 Single Counterexamples

We first consider the case where D contains a single path. Since we do not consider procedures yet, the nodes in D correspond to control points in the program (without any calling context). In this case, we can equivalently view D as a sequence of (intraprocedural) statements.

Consider the following SPDS with its equivalent program formulation:

$$\begin{array}{ll}
\langle 0\rangle \hookrightarrow \langle 1\rangle & (X' \wedge (Y' \leftrightarrow Y) \wedge (Z' \leftrightarrow Z)) \\
\langle 1\rangle \hookrightarrow \langle 2\rangle & ((X' \leftrightarrow X) \wedge (Y' \leftrightarrow X) \wedge (Z' \leftrightarrow Z)) \\
\langle 2\rangle \hookrightarrow \langle 3\rangle & ((\neg Y \wedge Z) \wedge (X' \leftrightarrow X) \wedge (Y' \leftrightarrow Y) \wedge (Z' \leftrightarrow Z))
\end{array}$$

```
0:  $X :=$ true
1:  $Y := X$
2:  if $(\neg Y \wedge Z)$ then
3:      error
```

Clearly, `error` is not reachable. However, if we check the initial abstraction that ignores data, we obtain the (unique) abstract counterexample trace $X :=$ **true**; $Y := X$; `assume`$(\neg Y \wedge Z)$. We demonstrate how by computing interpolants we can simultaneously show that the trace is spurious and find an explanation of why it is so. Renaming the variables in the trace yields the following formulas:

$$\begin{array}{lcll}
F_1 & \equiv & X_1 \wedge (Y_1 \leftrightarrow Y_0) \wedge (Z_1 \leftrightarrow Z_0) & // \ X := \textbf{true} \\
F_2 & \equiv & (X_2 \leftrightarrow X_1) \wedge (Y_2 \leftrightarrow X_1) \wedge (Z_2 \leftrightarrow Z_1) & // \ Y := X \\
F_3 & \equiv & (\neg Y_2 \wedge Z_2) \wedge (X_3 \leftrightarrow X_2) \wedge (Y_3 \leftrightarrow Y_2) \wedge (Z_3 \leftrightarrow Z_2) & // \ \texttt{assume}(\neg Y \wedge Z)
\end{array}$$

For instance, the variables with index 2 (X_2, Y_2 and Z_2) refer to the values of X, Y and Z after $X :=$ **true**; $Y := X$ has been executed, and before `assume`$(\neg Y \wedge Z)$ has been executed. The characteristic formula of the trace is $F_D \equiv F_1 \wedge F_2 \wedge F_3$. It is unsatisfiable if and only if the trace is spurious.

The procedures derived from Proposition 2.6 show that F_D is indeed unsatisfiable and yield the following strongest and weakest interpolants:

$$\begin{array}{lclcllcl} I_1 & = & SI(F^{..1}, F^{2..}) & \equiv & \exists\{Y_0, Z_0, Y_1\}.F_1 & & \equiv & X_1 \\ I_2 & = & SI(F^{..2}, F^{3..}) & \equiv & \exists\{X_1, Z_1\}.(SI(F^{..1}, F^{2..}) \wedge F_2) & & \equiv & (X_2 \wedge Y_2) \\ J_2 & = & WI(F^{..2}, F^{3..}) & \equiv & \forall\{X_3, Y_3, Z_3\}.\neg F_3 & & \equiv & (Y_2 \vee \neg Z_2) \\ J_1 & = & WI(F^{..1}, F^{2..}) & \equiv & \forall\{X_2, Y_2, Z_2\}.(F_2 \rightarrow WI(F^{..2}, F^{3..})) & & \equiv & (X_1 \vee \neg Z_1) \end{array}$$

Thus, the predicate P_n we are interested in at node n (where $n = 0, 1, 2, 3$), is an interpolant for the formula pair $(F^{..n}, F^{n+1..})$, which is in fact a predicate over variable values at n. For instance, the interpolants $SI(F^{..2}, F^{3..})$ and $WI(F^{..2}, F^{3..})$, or any other interpolant for this pair, can only contain logical variables common to $F^{..2}$ and $F^{3..}$, which must necessarily have index 2. These logical variables refer to the values of the program variables after the execution of $X := \textbf{true}; Y := X$ and before the execution of $\texttt{assume}(\neg Y \wedge Z)$. The reader may observe that the interpolant computation is, in fact, equivalent to a computation of strongest postconditions and weakest preconditions.

Fact 4.1 *Let $F_1 \wedge \ldots \wedge F_k$ be the (unsatisfiable) characteristic formula of a spurious trace consisting of statements $c_1; c_2; \ldots; c_k$, let $\{K_i\}$ be a family of interpolants satisfying the tracking property, and let P_i be the predicate over program variables obtained by removing the index i from all logical variables in K_i.*
Then $\{\textbf{true}\}c_1\{P_1\}c_2\{P_2\}\ldots\{P_{k-1}\}c_k\{\textbf{false}\}$ is a valid Hoare annotation.

Hence, interpolants satisfying the tracking property "explain" the infeasibility of a trace by providing Hoare annotations. In our example we obtain

$$\begin{array}{lllllllr} \{\textbf{true}\} & X := \textbf{true} & \{X\} & Y := X & \{X \wedge Y\} & \texttt{assume}(\neg Y \wedge Z) & \{\textbf{false}\} & (I_i), \\ \{\textbf{true}\} & X := \textbf{true} & \{X \vee \neg Z\} & Y := X & \{Y \vee \neg Z\} & \texttt{assume}(\neg Y \wedge Z) & \{\textbf{false}\} & (J_i). \end{array}$$

Notice that, by definition, we have $I_i \models J_i$; for instance, $X \wedge Y \models Y \vee \neg Z$.

In this example, conciliated interpolants provide a better explanation of infeasibility. The procedures of Proposition 2.8 guarantee the tracking property and lead to the Hoare annotation

$$\{\textbf{true}\} \quad X := \textbf{true} \quad \{X\} \quad Y := X \quad \{Y\} \quad \texttt{assume}(\neg Y \wedge Z) \quad \{\textbf{false}\}$$

Intuitively, a strongest interpolants at node n records *all* facts that are established by the path leading up to n; e.g. the strongest interpolant at node 2 is $X \wedge Y$. The weakest interpolant at n represents the disjunction of all conditions that make the trace infeasible if they hold at n; e.g. the weakest interpolant at node 2 is $Y \vee \neg Z$. The conciliated interpolant combines both aspects: it takes the facts that can be established at node n (the strongest interpolant) and filters out the information that is actually relevant for making the trace infeasible (the weakest interpolant); in the example, the conciliated interpolant at node 2 is Y.

4.2 Multiple Counterexamples

We now extend the techniques from Section 4.1 to the more general case where D contains multiple paths to the error. First, we adapt the construction of F_D. This is illustrated by the following formula, which represents the DAG in Figure 4. For every node in the DAG, we take the disjunction over the labels on its incoming arcs; e.g., at control point 4, where two branches of the DAG merge, we take the disjunction of the labels on the edges from nodes 2 and 2′.

$$\begin{array}{rll} F_D \equiv & (X_2 = X_1 \cdot (X_1 + 1)) & \wedge\ (Y_2 = Y_1) \\ \wedge & (X_3 = X_2) & \wedge\ (Y_3 = Y_2 \text{ odd}) \\ \wedge & (X_{2'} = X_3) & \wedge\ (Y_{2'} = Y_3 + 1) \\ \wedge & (((X_4 = X_2) & \wedge\ (Y_4 = Y_2 \text{ even})) \vee ((X_4 = X_{2'}) \wedge (Y_4 = Y_{2'} \text{ even}))) \\ \wedge & (X_{error} = X_4) & \wedge\ (Y_{error} = Y_4) \wedge (X_4 + Y_4 \text{ odd}). \end{array}$$

As before, D is spurious if and only if F_D is unsatisfiable. For a node n, let us define the *formula pair of* n as (F, G), where F is the formula corresponding to the DAG “above n” and G is the formula corresponding to the DAG “below n”. Then, our predicate P_n is an interpolant for the formula pair of n. In the example above, P_3 is an interpolant for the formula pair (F_3, G_3), where

$$\begin{array}{rcl} F_3 & \equiv & (X_2 = X_1 \cdot (X_1 + 1)) \wedge (Y_2 = Y_1) \wedge (X_3 = X_2) \wedge (Y_3 = Y_2 \text{ odd}), \\ G_3 & \equiv & (X_{2'} = X_3) \wedge (Y_{2'} = Y_3 + 1) \wedge (X_4 = X_{2'}) \wedge (Y_4 = Y_{2'} \text{ even}) \\ & & \wedge\ (X_{error} = X_4) \wedge (Y_{error} = Y_4) \wedge (X_4 + Y_4 \text{ odd}). \end{array}$$

It is easy to see that, in spurious DAGs, such formula pairs are unsatisfiable. By definition, only current variable values can occur in interpolants for those pairs, i.e., variable values with index 3 in the example above.

Strongest and weakest interpolants at each control point in D can be computed in a stepwise way as sketched in Props. 2.6 and 2.10.

In the example, the predicates in curly brackets in Figure 4 are weakest interpolants. Proposition 2.10 (ii) is used to compute the interpolant at point 2. Since the predicate computed at 1 turns out to be **true**, one can infer (cf. Proposition 2.6) that the DAG is spurious and the computed predicates are indeed interpolants. Strongest interpolants could be computed similarly. In that case, the DAG is spurious if the predicate at *error* is indeed **false**.

Thanks to the tracking property, the interpolants computed in this manner explain the infeasibility of the traces in the DAG. For instance, we have the valid Hoare triple $\{X \text{ even} \vee Y \text{ even}\}\ Y := Y + 1\ \{X \text{ even} \vee\ Y \text{ odd}\}$. Combined, for the whole DAG D we have the Hoare triple $\{\textbf{true}\}\ D\ \{\textbf{false}\}$, which is an alternative way to state the spuriousness of D.

Example 4.2 *DAGs can represent exponentially many counterexamples. Consider, for instance, the program in Figure 5. This program contains* $2^3 = 8$ *different paths to the error label, which are all shown in the DAG on the right side.*

In this example, a CEGAR scheme that considers only single counterexamples requires $2^3 - 1 = 7$ *refinement cycles, essentially one for each single path (assuming a simple predicate generation strategy such as weakest or strongest interpolants). This number grows*

```
bool A; bool B[3];

1:  $A :=$ true
2:  if $B[0]$ then
3:      skip
4:  else skip
5:  if $B[1]$ then
6:      skip
7:  else skip
8:  if $B[2]$ then
9:      skip
10: else skip
11: if $\neg A$ then error
```

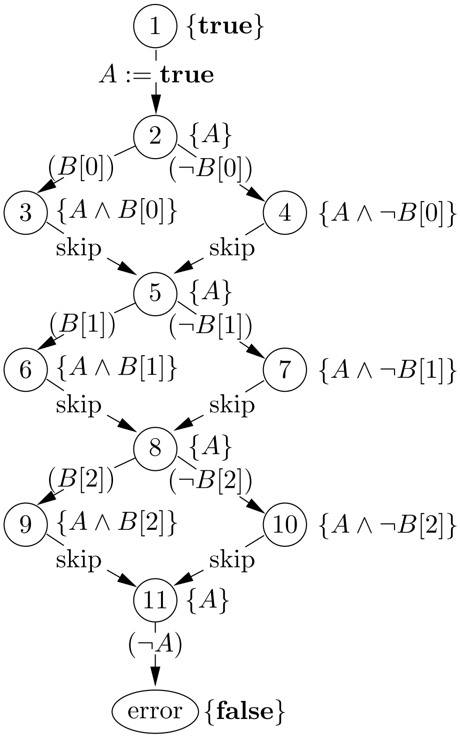


Figure 5. Program with many paths

exponentially with the number of "diamonds" in the DAG. The problem gets worse when such a program structure occurs as a part of a bigger program.

With the DAG of Figure 5, on the other hand, we can exclude all counterexamples in a single cycle. Intuitively, the strongest interpolant $\{A\}$ *at the control points* $2, 5, 8$ *and* 11 *is the "right" predicate: It is simple and powerful enough to rule out the whole DAG, because the different paths to the points* $5, 8$ *and* 11 *demonstrate that the values of the Boolean array* $B[\,]$ *are irrelevant and only the value of* A *matters.*

4.3 Programs with Procedures

We now discuss the case where the underlying SPDS represents a program with (possibly recursive) procedures. The nodes of D now represent control points of the program *plus calling context*, i.e., a stack of return addresses.

The construction of the characteristic formula F_D is the same as in Section 4.2. However, F_D now contains global and local variables. Local variables are saved during procedure calls and restored upon completion of a procedure. Thus, if we consider the formula pair (F, G) at a node n, where n is inside a callee, the local variables of the callers become part of the common variables of F and G and could occur in P_n. However, we believe that P_n should be independent of the calling context, for two reasons:

- To generate the abstract transition rules in a simple and efficient way (see Section 5), the predicate P_n should depend only on the data that is available in the concrete transition rules that lead into or out of n.

- Allowing local data from the callers to 'pollute' the abstract data space of the callee would severely impair the usefulness of the SPDS model, effectively 'flattening' the system into one that resembles a version where all procedures have been inlined.

In the following, we sketch the modifications that arise in this case. Our goal is to ensure that the predicates P_n at each node n are independent of the calling context and still satisfy the tracking property. For simplicity of the presentation we assume that there is a single global variable G and one local variable L in each procedure.

- For all nodes n, we generate a predicate $P_n(G_{in}, L_{in}, G, L)$ recording a relation between the global/local data G, L at n and the data G_{in}, L_{in} that was valid when entering the procedure that n belongs to. If n corresponds to the entry point of a procedure, we ensure $(G_{in} \leftrightarrow G) \wedge (L_{in} \leftrightarrow L) \models P_n$.

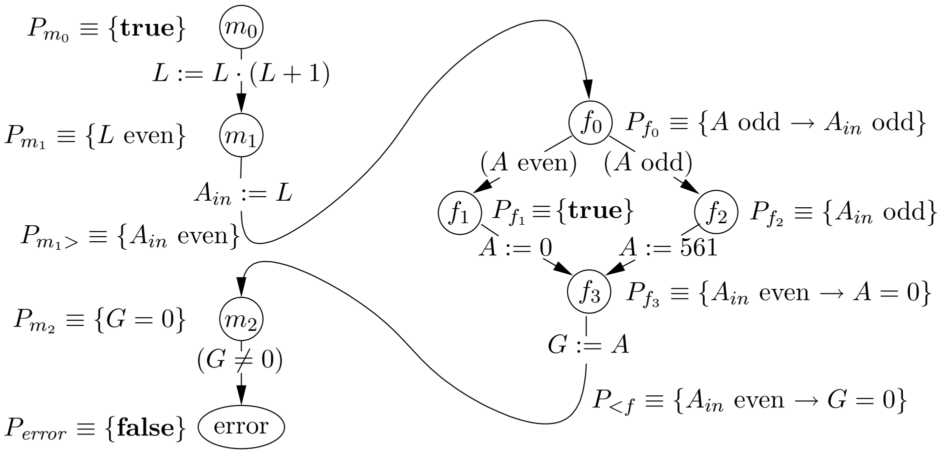
- If an edge from n to n' is labeled by an intraprocedural rule R, we ensure $P_n \wedge R \models P_{n'}$, preserving the tracking property.

- If an edge from node n is labeled by a transition rule $Push(G, L, G', L', L'')$ (modeling a call), we generate an interpolant $P_{n>}(G_{in}, L_{in}, G', L', L'')$ s.t. $P_n \wedge Push \models P_{n>}$. Thus, $P_{n>}$ contains information about the arguments given to the callee (G', L') and the saved local data (L'').

- If an edge from node n' is labeled by a transition rule $Pop(G, L, G')$ (a return statement) and `f` was the called procedure, we generate an interpolant $P_{<f}(G_{in}, L_{in}, G')$

s.t. $P_{n'} \wedge Pop \models P_{<f}$. The predicate $P_{<f}$ is effectively an input-output relation of the called procedure `f` or, in other words, a predicate that argues about the effect of `f`. So, if n'' is the target node of the edge and n is the node from which the call took place, we ensure that $P_{n>} \wedge P_{<f} \models P_{n''}$.

Figure 6 gives an example for a (spurious) counterexample DAG to the SPDS in Figure 3, which contains a procedure call. The left-hand side shows the control flow in the procedure `m`, which is interrupted by a call to a function `f`, whose control flow is shown on the right. The predicates associated with the nodes are the weakest interpolants for our example.



$P_{m_0} \equiv \{\mathbf{true}\}$
m_0
$L := L \cdot (L+1)$
$P_{m_1} \equiv \{L \text{ even}\}$
m_1
$A_{in} := L$
$P_{m_1>} \equiv \{A_{in} \text{ even}\}$
$P_{m_2} \equiv \{G = 0\}$
m_2
$(G \neq 0)$
$P_{error} \equiv \{\mathbf{false}\}$
error
f_0
$P_{f_0} \equiv \{A \text{ odd} \rightarrow A_{in} \text{ odd}\}$
$(A \text{ even})$
$(A \text{ odd})$
f_1
$P_{f_1} \equiv \{\mathbf{true}\}$
f_2
$P_{f_2} \equiv \{A_{in} \text{ odd}\}$
$A := 0$
$A := 561$
f_3
$P_{f_3} \equiv \{A_{in} \text{ even} \rightarrow A = 0\}$
$G := A$
$P_{<f} \equiv \{A_{in} \text{ even} \rightarrow G = 0\}$

Figure 6. An example for a counterexample DAG with procedure call

To explain how the predicates are computed in the general case, we consider a generic procedure call/return pair, see Figure 7. A procedure `m` calls a procedure `f` and is resumed after the termination of `f`.

In order to simplify our description, we assume that the variables that the rules might depend on (they are put in parentheses) actually occur there. Table 1 shows the computation of strongest and weakest interpolants for the predicates P_x.

Most of the top-down computation of strongest interpolants (left-hand side of Table 1) is quite straightforward and resembles Proposition 2.6. Observe that the predicate S_{f_0} simply states that the local and global variable values at the beginning of `f` equal the local and global variable values at point f_0. This paves the way for the following predicates $S_{f_1}, S_{f_2}, \ldots$ that keep track of the relation between the variable values at the beginning of `f` $(G_{f_{in}}, L_{f_{in}})$ and the current variable values. The predicate $S_{<f}$ finally captures the input-output behavior of `f`. Notice that $S_{m_{i+1}}$, the strongest interpolant after the execution of `f`, is computed by combining S_{m_i} (the strongest interpolant before `f`) with the procedure effect $S_{<f}$.

The bottom-up computation scheme of weakest interpolants (right-hand side of Table 1) requires some more explanation. Consider the predicate $W_{m_i>}$ and assume that we have already computed $W_{m_{i+1}}$ in our bottom-up computation scheme. Earlier, we noted that we want to guarantee $W_{m_i>} \wedge P_{<f} \models W_{m_{i+1}}$, no matter what the predicate $P_{<f}$ may be. The

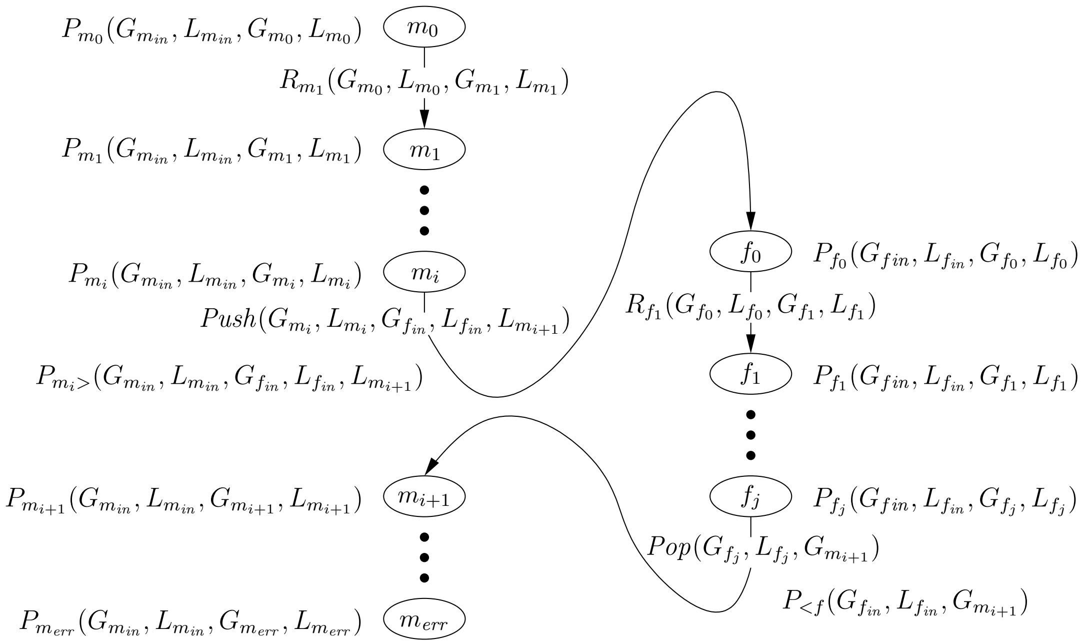

$P_{m_0}(G_{m_{in}}, L_{m_{in}}, G_{m_0}, L_{m_0})$
m_0
$R_{m_1}(G_{m_0}, L_{m_0}, G_{m_1}, L_{m_1})$
$P_{m_1}(G_{m_{in}}, L_{m_{in}}, G_{m_1}, L_{m_1})$
m_1
$P_{m_i}(G_{m_{in}}, L_{m_{in}}, G_{m_i}, L_{m_i})$
m_i
$Push(G_{m_i}, L_{m_i}, G_{f_{in}}, L_{f_{in}}, L_{m_{i+1}})$
$P_{m_i>}(G_{m_{in}}, L_{m_{in}}, G_{f_{in}}, L_{f_{in}}, L_{m_{i+1}})$
f_0
$P_{f_0}(G_{fin}, L_{f_{in}}, G_{f_0}, L_{f_0})$
$R_{f_1}(G_{f_0}, L_{f_0}, G_{f_1}, L_{f_1})$
f_1
$P_{f_1}(G_{fin}, L_{f_{in}}, G_{f_1}, L_{f_1})$
$P_{m_{i+1}}(G_{m_{in}}, L_{m_{in}}, G_{m_{i+1}}, L_{m_{i+1}})$
m_{i+1}
f_j
$P_{f_j}(G_{fin}, L_{f_{in}}, G_{f_j}, L_{f_j})$
$Pop(G_{f_j}, L_{f_j}, G_{m_{i+1}})$
$P_{<f}(G_{f_{in}}, L_{f_{in}}, G_{m_{i+1}})$
$P_{m_{err}}(G_{m_{in}}, L_{m_{in}}, G_{m_{err}}, L_{m_{err}})$
m_{err}

Figure 7. A counterexample DAG with procedure call

weakest formula over $(G_{m_{in}}, L_{m_{in}}, G_{f_{in}}, L_{f_{in}}, L_{m_{i+1}})$ that satisfies this implication is

$$W_{m_i>} \equiv \forall\{G_{m_{i+1}}\}.(P_{<f} \rightarrow W_{m_{i+1}}) \,.$$

Now one can see that a stronger $P_{<f}$ will make $W_{m_i>}$ weaker. Since we wish $W_{m_i>}$ to be as weak as possible, we choose for $P_{<f}$ the *strongest* interpolant $S_{<f}$.

Once $W_{m_i>}$ is fixed, the predicate $W_{<f}$ can be computed. The weakest formula $W_{<f}$ over $(G_{f_{in}}, L_{f_{in}}, G_{m_{i+1}})$ that satisfies the implication $W_{m_i>} \wedge W_{<f} \models W_{m_{i+1}}$ is

$$W_{<f} \equiv \forall\{G_{m_{in}}, L_{m_{in}}, L_{m_{i+1}}\}.(W_{m_i>} \rightarrow W_{m_{i+1}}) \,,$$

Table 1. Computation of strongest and weakest interpolants

	strongest interpolant S	weakest interpolant W
S_{m_0}/W_{m_0}	**true**	$\forall\{G_{m_1}, L_{m_1}\}.(R_{m_1} \rightarrow W_{m_1})$
S_{m_1}/W_{m_1}	$\exists\{G_{m_0}, L_{m_0}\}.(S_{m_0} \wedge R_{m_1})$	bottom-up
S_{m_i}/W_{m_i}	top-down	$\forall\{G_{f_{in}}, L_{f_{in}}, L_{m_{i+1}}\}.(Push \rightarrow W_{m_i>})$
$S_{m_i>}/W_{m_i>}$	$\exists\{G_{m_i}, L_{m_i}\}.(S_{m_i} \wedge\ Push)$	$\forall\{G_{m_{i+1}}\}.(S_{<f} \rightarrow W_{m_{i+1}})$
S_{f_0}/W_{f_0}	$(G_{f_{in}} \leftrightarrow G_{f_0}) \wedge (L_{f_{in}} \leftrightarrow L_{f_0})$	$\forall\{G_{f_1}, L_{f_1}\}.(R_{f_1} \rightarrow W_{f_1})$
S_{f_1}/W_{f_1}	$\exists\{G_{f_0}, L_{f_0}\}.(S_{f_0} \wedge R_{f_1})$	bottom-up
S_{f_j}/W_{f_j}	top-down	$\forall\{G_{m_{i+1}}\}.(Pop \rightarrow W_{<f})$
$S_{<f}/W_{<f}$	$\exists\{G_{f_j}, L_{f_j}\}.(S_{f_j} \wedge\ Pop)$	$\forall\{G_{m_{in}}, L_{m_{in}}, L_{m_{i+1}}\}.(W_{m_i>} \rightarrow W_{m_{i+1}})$
$S_{m_{i+1}}/W_{m_{i+1}}$	$\exists\{G_{f_{in}}, L_{f_{in}}\}.(S_{m_i>} \wedge S_{<f})$	bottom-up
S_{m_n}/W_{m_n}	top-down	**false**

```
bool $G$;

procedure main {
    $G :=$ true
    level$_1$()
    level$_1$()
    if ($G ==$ false) then
        error
}

procedure level$_1$() {
    level$_2$()
    level$_2$()
}
⋮
procedure level$_{n-1}$() {
    level$_n$()
    level$_n$()
}
procedure level$_n$() {
    if ($G ==$ true) then $G :=$ false
    else $G :=$ true
}
```

Figure 8. Program whose first abstraction includes an exponentially long spurious counterexample

as stated in Table 1.

On a more intuitive level, we use the most precise description of f's effect available to us, namely the strongest interpolant $S_{<f}$, to compute $W_{m_i>}$ from $W_{m_{i+1}}$. Thus we get the weakest $W_{m_i>}$ possible. Once $W_{m_i>}$ is known, we can weaken $S_{<f}$ to $W_{<f}$ by overapproximating f's effect in terms of the given counterexample DAG: We need not say more about $W_{<f}$ than that it transforms $W_{m_i>}$ to $W_{m_{i+1}}$.

It is straightforward to extend this scheme so as to handle unfinished procedure calls, e.g. if the error label is reached before a callee completes its execution. In this case, two simplifications can be applied:

- We need not care about the local variables of the caller after the call because they are irrelevant for counterexamples.
- We need not track an input-output relation of the callee because unfinished procedures produce no output.

Conceptually, we inline the procedure call in this case and treat the push rule like an ordinary intraprocedural statement: the local variables of the caller (the variable $L_{m_{i+1}}$ in Figure 7) are quantified away, and the input-output relation of the callee is omitted.

Our implementation improves the efficiency of predicate generation by working directly on the witness graphs discussed in [14]. This data structure represents counterexample DAGs in a compact way by reusing nodes from procedures that are called multiple times. For example, consider the program in Figure 8 (simplified from [15]), whose first abstraction contains an exponentially (in n) long spurious counterexample.

In [15] and [14] it is discussed how to model check such a program in linear time. Moped constructs the witness graph while it model checks, so the resulting representation of the abstract counterexample has only linear size. We can take advantage of this representation and compute only one predicate for each node in the compressed counterexample DAG. This leads to a linear number of predicates in the example above. In the following, we sketch how this is accomplished.

The strongest interpolants in a callee do not depend on the calling context (see $S_{f_0}, S_{f_1}, S_{f_j}$ in Table 1), so it is easy to compute those predicates only once for all invocations of the procedure. In contrast, the weakest interpolants in a callee do depend on the context: In Table 1, the predicates $W_{f_0}, W_{f_1}, W_{f_j}$ depend indirectly on the previously computed $W_{m_{i+1}}$. In order to compute those predicates only once for each node in the witness graph, we first compute the "effect" of `f` relative to the context. This procedure effect is given by $W_{<f}$ for each invocation that uses the corresponding witness node. The procedure effect for *all* those invocations can then be obtained by taking the conjunction of all $W_{<f}$. The resulting predicate expresses the effect of *any* of those invocations of `f`.

This treatment of multiple branches in a counterexample is quite analogous to the one of Section 4.2. Both times we use a disjunction or a conjunction to summarize multiple branches according to Proposition 2.10. Note that this would not work for recursive programs, if the witness graph contained cycles. Fortunately, witness graphs are always DAGs. If a procedure calls itself in a counterexample, the two invocations are represented by distinct nodes in the witness graph [14].

We conclude that the witness graph structure gives us two *different* exponential savings. In Section 4.2, we gave a counterexample DAG that contains an exponential *number* of counterexamples. In this section, we mentioned that counterexamples of exponential *length* can be compressed by reusing procedure invocations.

5. Computing the Abstract SPDS

In each CEGAR cycle, we derive predicates to refine our abstraction. In the methods of Section 4, each predicate naturally belongs to a control point. Thus, as in [7], we maintain for each control point a list of predicates that are useful there. In the following we explain how, given a concrete SPDS and a predicate list, one can compute an (overapproximating) abstract SPDS.

5.1 An Example

Consider the example SPDS of Section 4.1 which contains no procedure calls. We derived conciliated interpolants that explain the infeasibility of the error trace. At each control point, we now associate each predicate (except for **true** and **false**) with a Boolean variable that reflects the truth of the predicate: $[1] := l_1 \leftrightarrow X$ and $[2] := l_2 \leftrightarrow Y$.

For the computation of the abstract rules, we use existential abstraction. For instance, the concrete BDD $R \equiv (X' = X) \wedge (Y' = X) \wedge (Z' = Z)$ of the SPDS rule $\langle 1 \rangle \hookrightarrow \langle 2 \rangle\ (R)$ is replaced by an "abstract" BDD

$\exists\{X, Y, Z, X', Y', Z'\}.\big((l_2 \leftrightarrow X) \wedge ((X' \leftrightarrow X) \wedge (Y' \leftrightarrow X) \wedge (Z' \leftrightarrow Z)) \wedge (l_3' \leftrightarrow Y')\big) \equiv l_2' \leftrightarrow l_1.$

The number of abstract variables can be reduced because we track the predicates only at the control points where they were derived. In our example, we have only one predicate per control point. Therefore, one abstract Boolean variable suffices for the abstract SPDS:

$$
\begin{array}{lll}
\langle 0\rangle \hookrightarrow \langle 1\rangle & (l') \\
\langle 1\rangle \hookrightarrow \langle 2\rangle & (l' \leftrightarrow l) \\
\langle 2\rangle \hookrightarrow \langle 3\rangle & (\neg l)
\end{array}
$$

```
0:  $l := \textbf{true}$
1:  skip
2:  if $\neg l$ then
3:      error
```

The error label is no longer reachable in the abstract program. This is due to the fact that the Hoare annotation of a concrete program can be abstractly translated:

$$
\begin{array}{llllllll}
\{\textbf{true}\} & X := \textbf{true} & \{X\} & Y := X & \{Y\} & \texttt{assume}(\neg Y \wedge Z) & \{\textbf{false}\} & \text{translates into} \\
\{\textbf{true}\} & l := \textbf{true} & \{l\} & \texttt{skip} & \{l\} & \texttt{assume}(\neg l) & \{\textbf{false}\}. &
\end{array}
$$

Hence, if the predicates that explain the infeasibility of a trace are added to the program by means of an existential abstraction as above, this spurious trace is excluded.

5.2 The General Case

We now discuss how to compute an abstract SPDS. The idea, as seen in the example, is to introduce Boolean variables in the abstract SPDS that track the truth of the predicates.

In order to simplify the presentation, we show the construction for the case where there is only one predicate per control point. Hence, we may reuse the notation of Figure 7 and Table 1. We also assume again that there is a single global variable G and one local variable L at each control point. The techniques can easily be generalized to eliminate those assumptions.

As sketched in Section 5.1, we compute *concretizations* for each control point. These concretizations map the abstract variables (in small letters) to the corresponding predicates over the concrete variables (in capital letters). We also compute concretizations for some additional control points that occur only in the abstract SPDS, see below. For the program in Fig. 7, we compute the following concretizations:

$$
\begin{array}{rcccl}
[m_0] & \equiv & l_{m_0} & \leftrightarrow & P_{m_0}(G_{m_{in}}, L_{m_{in}}, G_{m_0}, L_{m_0}) \\
[m_1] & \equiv & l_{m_1} & \leftrightarrow & P_{m_1}(G_{m_{in}}, L_{m_{in}}, G_{m_1}, L_{m_1}) \\
[m_i] & \equiv & l_{m_i} & \leftrightarrow & P_{m_i}(G_{m_{in}}, L_{m_{in}}, G_{m_i}, L_{m_i}) \\
[m_i\!>] & \equiv & l_{m_i>} & \leftrightarrow & P_{m_i>}(G_{m_{in}}, L_{m_{in}}, G_{f_{in}}, L_{f_{in}}, L_{m_{i+1}}) \\
[f_0] & \equiv & l_{f_0} & \leftrightarrow & P_{f_0}(G_{f_{in}}, L_{f_{in}}, G_{f_0}, L_{f_0}) \\
[f_j] & \equiv & l_{f_j} & \leftrightarrow & P_{f_j}(G_{f_{in}}, L_{f_{in}}, G_{f_j}, L_{f_j}) \\
[<\!f] & \equiv & g & \leftrightarrow & P_{<f}(G_{f_{in}}, L_{f_{in}}, G_{m_{i+1}}) \\
[m_{i+1}] & \equiv & l_{m_{i+1}} & \leftrightarrow & P_{m_{i+1}}(G_{m_{in}}, L_{m_{in}}, G_{m_{i+1}}, L_{m_{i+1}})
\end{array}
$$

If the predicate list at control point n consists of more than one predicate, a concretization $[n]$ becomes the conjunction of more than one equivalence.

The abstract variables l_x may all be local. In contrast, the effect of a called procedure (the predicate $P_{<f}$) must be captured in an abstract global variable g, because it needs to be inspected later by the caller in order to incorporate the procedure effect into its local abstract variable values. Notice that we abuse Fig. 7 here in the sense that the abstract SPDS does not depend on a particular counterexample DAG, but only on the computed predicates which happen to be shown in Fig. 7.

For each type of rule in the concrete SPDS (intraprocedural rule, push rule, pop rule), we now provide the details for generating the corresponding abstract rules. The general idea is to take the concrete BDD R, conjoin it with the concretizations of the predicates of interest both before and after the execution of the rule, and finally to existentially abstract away the concrete variables. The latter operation is denoted below by Abs, i.e. $Abs(R)$ existentially abstracts away all copies of G and L occurring in R.

A concrete intraprocedural rule

$$\langle m_0 \rangle \hookrightarrow \langle m_1 \rangle \quad (R_{m_1}(G_{m_0}, L_{m_0}, G_{m_1}, L_{m_1}))$$

is replaced by the abstract rule

$$\langle m_0 \rangle \hookrightarrow \langle m_1 \rangle \quad (r_{m_1}(l_{m_0}, l_{m_1})),$$

where

$$r_{m_1} \equiv Abs\big([m_0] \wedge R_{m_1} \wedge [m_1]\big) \ .$$

This case was illustrated in Section 5.1. However, there we neglected the relation to the input variables $G_{m_{in}}$ and $L_{m_{in}}$.

A concrete push rule

$$\langle m_i \rangle \hookrightarrow \langle f_0, m_{i+1} \rangle \quad (Push(G_{m_i}, L_{m_i}, G_{f_{in}}, L_{f_{in}}, L_{m_{i+1}}))$$

is replaced by two abstract rules:

$$\langle m_i \rangle \hookrightarrow \langle f_0, m_i{>} \rangle \quad (push(l_{m_i}, l_{f_0}, l_{m_i>}))$$

and

$$\langle m_i{>} \rangle \hookrightarrow \langle m_{i+1} \rangle \quad (eval(l_{m_i>}, g, l_{m_{i+1}}))\ ,$$

where $m_i{>}$ is a new control point,

$$push \equiv Abs\big([m_i] \wedge Push \wedge (G_{f_{in}} \leftrightarrow G_{f_0}) \wedge (L_{f_{in}} \leftrightarrow L_{f_0}) \wedge [f_0] \wedge [m_i{>}]\big)\ ,$$

and

$$eval \equiv Abs\big([m_i{>}] \wedge [{<}f] \wedge [m_{i+1}]\big)\ .$$

We introduce the new control point $m_i{>}$ because we need an additional evaluation step in order to set the value of $l_{m_{i+1}}$ correctly: The status directly after the procedure call, saved in $l_{m_i>}$, is combined in *eval* with the procedure effect, saved in g.

Finally, a concrete pop rule

$$\langle f_j \rangle \hookrightarrow \langle\rangle \quad (Pop(G_{f_j}, L_{f_j}, G_{m_{i+1}}))$$

is replaced by the abstract rule

$$\langle f_j \rangle \hookrightarrow \langle\rangle \quad (pop(l_{f_j}, g))\ ,$$

where

$$pop \equiv Abs\big([f_j] \wedge Pop \wedge [{<}f]\big)\ .$$

As mentioned in Section 5.1, we can save variables in a straightforward way, since each local variable l_x is relevant at only one control point. We did not spend much effort to optimize the assignment of predicates to BDD variables, as model checking the abstract system with Moped did not turn out to be costly.

6. Case Studies

We have implemented the ideas of this paper in an extension of Moped, in order to decrease resources needed for model checking SPDSs. Moped accepts multiple input languages including a subset of Java [17, 16].

Sections 6.1–6.3 demonstrate the benefits of abstraction refinement with conciliated interpolants. For these examples, a comparison of our program with existing CEGAR tools did not seem appropriate because the assumptions of tools like BLAST and SLAM (infinite variable ranges, theorem provers) differ significantly from ours (finite variable ranges). However, in Sections 6.4 and 6.5 we provide a comparison with BLAST to illustrate a bad asymptotic behavior if some techniques described in this paper are not used. In Section 6.6 we describe unsuccessful experiments.

6.1 Locking Example

Figure 9 shows an example of a program where CEGAR clearly pays off, especially when the number of bits for the integer variables ("bit width") is increased. The "*"-sign stands for a nondeterministic value. We want to check the fact that the assertions in the program always hold. This property is actually independent of the integer variables. Table 2 shows performance results (on an Intel Xeon CPU 2.40GHz and using a bit width of 8).

```
struct file {
    bool locked;
    int pos;
};
open(file f) {
    assert(¬f.locked);
    f.locked = true;
    f.pos = 0;
}
close(file f) {
    assert(f.locked ∨
            f.pos==0);
    f.locked = false;
}
```

```
rw(file f) {
    assert(f.locked ∨ f.pos==0);
    f.pos = f.pos + 1;
}
main() {
    file f1,f2;
    f1.locked = f2.locked = false;
    open(f1);
    while(*) {
        open(f2);
        while(*) { rw(f2); rw(f1); }
        close(f2);
    }
    close(f1);
}
```

Figure 9. Locking example (pseudo code)

Table 2. Results of different Moped versions applied on the locking example

	time/s	memory/BDD nodes	# cycles	# gl. var.	# loc. var.
w/o CEGAR	460	440482	n/a	n/a	n/a
weakest interp.	0.43	89936	14	13	6
concil. interp.	0.29	80738	10	10	7

Moped without CEGAR needs exponential time in the bit width. On the other hand, using weakest or conciliated interpolants, our CEGAR scheme automatically abstracts from the integers and proves the assertions in constantly many refinement cycles. The number of global and local variables in the final abstract program (containing no spurious error traces anymore) is also shown in the table and is also independent of the bit width. Time and memory consumption of the abstract versions grow modestly with the bit width. Conciliated interpolants have the best performance because the predicate simplification allows them to "discover" that the `f.pos` fields are irrelevant to the property.

6.2 Equality Tracking Example

Figure 10 shows another example highlighting the possible benefit of conciliated interpolants. The `error` label is not reachable because `X` = `Y` holds. The idea of this example is that the variable `Z` is irrelevant for the property to be checked. However, the weakest-interpolants heuristic cannot discover this, as suggested by Table 3. The array `Q` simply provides for a large state space. We used a bit width of 3.

```
int X, Y, Z;
foo() {
    int P;
    int Q[8];
    while(*) {
        if(*) {
            P = * % 8;  /* any number in {0,...,7} */
            Q[P] = Q[P] + 1;
        } else {
            Z = *;  /* any number */
            X = X + P;
            Y = Y + P;
        }
    }
}

main() {
    X = 0;
    Y = 0;
    foo();
    if (X ≠ Y)
        if (Z == 0)
            error;
}
```

Figure 10. Equality tracking example (pseudo code)

Table 3. Results of different Moped versions applied on the equality tracking example

	time/s	memory/BDD nodes	# cycles	# gl. var.	# loc. var.
w/o CEGAR	25	1723092	n/a	n/a	n/a
weakest interp.	3.0	556990	8	3	6
concil. interp.	0.1	44968	3	1	1
strongest interp.	> 3600	?	>100	?	?

Notice that abstraction with strongest interpolants does not work well here. We often observed that and do not recommend this heuristic. The advantage of conciliated interpolants can be arbitrarily increased by increasing the size of the problem, e.g. by extending the array `Q`. On the other hand, conciliated interpolants are not always better than weakest interpolants. For example, if the statements involving `Z` are eliminated from Figure 10, then weakest and conciliated interpolants coincide and both prove the property in 3 refinement cycles and in less than 0.1 seconds. In practice, conciliated interpolants are never worse than weakest interpolants because the additional resources required to compute conciliated interpolants do not form a bottleneck. We therefore use conciliated interpolants as our standard heuristic.

6.3 LinkedList Example

Abstraction can also be useful in positive instances (where the error label is reachable) and in larger programs. As an example, we took Java code for the class `LinkedList` from a textbook on data structures [18] and modified only the `main` method, simulating a user who accesses class methods randomly:

```
public class LinkedList { ···
   private ListNode header;
   public static void main (String[] args) {
      LinkedList l = new LinkedList();
      while (NONDET())
         if (NONDET()) l.insert(null, l.zeroth());
         else l.remove(null);
      assert(l.header == null);
   }
}
```

The assertion to be checked is not valid in the class implementation.[4] This reachability problem is scalable, as the size of jMoped's heap representation is adjustable. We compared Moped with and without CEGAR on different problem sizes. We used 5 bits per pointer variable in all cases. Conciliated interpolants were used in the cases with CEGAR. A refinement was not necessary. Table 4 shows the results.

Table 4. Results of different Moped versions applied to the LinkedList example

	without CEGAR		with CEGAR	
heap size/bits	time/s	memory/BDD nodes	time/s	memory/BDD nodes
40	13	$2.8 \cdot 10^6$	23	$5.7 \cdot 10^6$
50	22	$3.3 \cdot 10^6$	28	$6.9 \cdot 10^6$
60	36	$3.7 \cdot 10^6$	33	$7.8 \cdot 10^6$
70	86	$4.9 \cdot 10^6$	31	$9.1 \cdot 10^6$
80	437	$7.0 \cdot 10^6$	36	$10.9 \cdot 10^6$
90	1474	$12.1 \cdot 10^6$	44	$12.9 \cdot 10^6$
100	4980	$24.8 \cdot 10^6$	51	$15.0 \cdot 10^6$

[4] However, we did not discover a bug here. The class implementation is correct.

Moped with CEGAR outperforms Moped without CEGAR with growing heap sizes. Table 4 suggests an exponential increase in runtime for the non-abstracted version, whereas the CEGAR version seems to grow linearly in the heap size. Memory consumption is comparable in both versions; the non-abstracted version has some advantage for smaller heap sizes.

6.4 DAG Example

To show the usefulness of excluding a whole DAG of counterexamples in one cycle, we slightly extended the program of Figure 5 (page 41), see Figure 11.

```
bool B[3];
bool C[3];
main() {
   B[0] = *; B[1] = *; B[2] = *;
   if (B[0] == false) C[0] = false; else C[0] = B[0];
   if (B[1] == false) C[1] = false; else C[1] = B[1];
   if (B[2] == false) C[2] = false; else C[2] = B[2];
   if (B[0] ≠ C[0]) error;
   if (B[1] ≠ C[1]) error;
   if (B[2] ≠ C[2]) error;
}
```

Figure 11. DAG example for $n = 3$ (pseudo code)

Notice that the program shown in Figure 11 can be generalized from $n = 3$ (the size of the arrays) to arbitrary n in a straightforward way. We ran both Moped with CEGAR and BLAST (version 3.0)[5.] on this code. Table 5 shows the results.

Table 5. Results of Moped with CEGAR and BLAST applied to the DAG example

	Moped with CEGAR	BLAST	
n	time/s	time/s	Nb iterations of reachability
1	0.02	0.7	44
2	0.02	1.1	118
3	0.02	2.1	260
4	0.03	5.0	542
5	0.03	12.3	1124
6	0.04	31.5	2358
7	0.05	86.2	5012
8	0.07	221.0	10750

The results suggest that BLAST needs exponential time in n. The exponential behavior of BLAST is further suggested by the “Nb iterations of reachability” number from BLAST’s

[5.] In all of our BLAST experiments we used the options `-cf -craig 1 -predH 7 -alias bdd` which were recommended to us by the authors of BLAST.

output. Moped with CEGAR is much faster. In particular, it always needs exactly one refinement, because the full counterexample DAG (containing 2^n spurious counterexamples) is already obtained in the very first abstraction. This DAG is completely excluded by the first refinement.

6.5 Level Example

We ran both Moped with CEGAR and BLAST (version 3.0) on the code of Figure 8 (page 45) and on some variation of this code, see below. Table 6 shows the results.

Table 6. Results of Moped with CEGAR and BLAST applied to the level example

	Moped with CEGAR	BLAST on initial code		BLAST on changed code	
			Nb iterations		Nb iterations
n	time/s	time/s	of reachability	time/s	of reachability
1	0.02	0.6	46	0.8	69
2	0.02	0.8	70	1.0	105
3	0.02	1.1	88	1.4	132
4	0.02	1.4	106	1.8	159
5	0.02	1.5	124	3.8	277
6	0.02	1.9	142	9.7	418
7	0.02	2.6	160	25.7	583
8	0.02	4.0	178	77.9	800
9	0.02	6.8	196	291.3	985

Moped with CEGAR is very fast on this example (< 0.1 s for all $n < 10$). According to the explanation from the end of Section 4, the runtime grows linearly with n. However, this growth is not apparent from the table, as n is too small. Independently from n, Moped with CEGAR needs exactly one refinement to rule out all spurious counterexamples.

We also ran Moped on a modified example in which we replaced the code of procedure $\texttt{level}_n\texttt{()}$

```
procedure $\texttt{level}_n$() {
    if ($G ==$ true) then $G :=$ false
    else $G :=$ true
}
```

by

```
procedure $\texttt{level}_n$() {
    if ($G ==$ false) then $G :=$ true
    else $G :=$ false
}
```

and left all other code unchanged. The results remained the same, since Moped is insensitive to the code modification because both code fragments lead to the same SPDS rule with the same BDD.

BLAST also checks the initial example in linear time (see columns 3 and 4), where the "Nb iterations of reachability" number from BLAST's output grows only linearly. On the other hand, when we ran BLAST on the modified example (see columns 5 and 6), it seemed to need exponential time in n to verify the code, where the number of iterations in BLAST grows super-linearly. This suggests that the predicate generation of BLAST is not stable with respect to this seemingly trivial modification, whereas the use of BDDs causes Moped to remain stable.

6.6 Discussion of Unsuccessful Experiments

With our predicate generation heuristics, CEGAR does not always pay off. This is usually the case when properties are checked whose validity depends on the correctness of large portions of the code working on large data. For instance, CEGAR does not turn out to be useful in order to verify that a Quicksort program indeed correctly sorts an array of numbers. In this case, the abstraction refinement loop essentially restores the original non-abstracted program, because our predicate generation heuristics fail to "understand" how Quicksort works. So the CEGAR cycle considers exponentially many permutations of the array to be sorted. The "raw" version of Moped (without CEGAR) performs much better here. The situation for other sorting algorithms is similar.

In general terms, the abstracted version is superior if the validity of a property can be explained by predicates that can be found by our interpolation methods. If it is not clear whether this is the case, we recommend to try Moped both with and without CEGAR. The performance may be extremely different.

7. Conclusions

While Craig interpolation has previously been used for CEGAR together with SAT solvers and theorem provers, we found that it is also useful to enhance a BDD-based model checker. Strongest and weakest interpolants form a framework inside which heuristics can be applied to find *good* predicates, e.g. conciliated interpolants. The number of refinement cycles often depends crucially on the quality of the derived predicates.

BDD-based model checkers record how program states can be reached in order to report possible counterexamples. This information can be exploited by a CEGAR scheme to exclude multiple counterexamples at the same time. This can save exponentially (in the size of the DAG) many refinement cycles.

Our CEGAR scheme can achieve large savings, especially if the property to be checked is much simpler than the full functionality of the program. For future research, we plan to further improve predicate generation heuristics. Possibilities include an adapted form of lazy abstraction [8] and the incorporation of dataflow information to detect relevant counterexample parts [10].

Acknowledgments

We thank Dejvuth Suwimonteerabuth for his great support with jMoped and Andreas Holzer for installing BLAST. We also thank Grégory Théoduloz for his help on using BLAST. Finally, we thank the anonymous referees for their valuable comments that helped to improve this paper significantly.

References

[1] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01*, LNCS 2057, pages 103–122, 2001.

[2] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE Computer Society Press, 2003.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV'00*, LNCS 1855, pages 154–169. Springer, 2000.

[4] W. Craig. Linear reasoning. A new form of the Herbrand-Genzen theorem. *Journal of Symbolic Logic*, **22**:250–268, 1957.

[5] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In *Proceedings of TACAS 2006*, **3920** of *LNCS*, pages 489–503, 2006.

[6] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *Proceedings of TACAS 2003*, LNCS 2619, pages 176–191. Springer, 2003.

[7] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proc. POPL'04*, pages 232–244. ACM Press, 2004.

[8] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL'02*, pages 58–70. ACM Press, 2002.

[9] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Proc. DAC'05*, pages 445–450. ACM Press, 2005.

[10] R. Jhala and R. Majumdar. Path slicing. In *Proc. of PLDI '05*, pages 38–47. ACM, 2005.

[11] S. Kiefer. Abstraction refinement for pushdown systems. Master's thesis, Universität Stuttgart, 2005. `http://www.fmi.uni-stuttgart.de/szs/publications/info/kiefersn.Kie05.shtml`.

[12] K.L. McMillan. Interpolation and SAT-based Model Checking. In *Proc. CAV'03*, LNCS 2725, pages 1–13. Springer, 2003.

[13] K.L. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of TACAS 2005*, LNCS 3440, pages 1–12. Springer, 2005.

[14] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, **58**(1–2):206–263, October 2005. Special Issue on the Static Analysis Symposium 2003.

[15] S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, TU München, 2002.

[16] D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jMoped: A test environment for Java programs. In *Proc. CAV'07*, LNCS 4590, pages 164–167, 2007.

[17] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *Proceedings of TACAS 2005*, LNCS 3440, pages 541–545. Springer, 2005.

[18] M.A. Weiss. *Data Structures and Algorithm Analysis in Java.* Addison-Wesley, 1998.