Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems *

Javier Esparza, Stefan Kiefer, and Stefan Schwoon

Institute for Formal Methods in Computer Science, University of Stuttgart
{esparza,kiefersn,schwoosn}@informatik.uni-stuttgart.de

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 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.

1 Introduction

CEGAR is a powerful tool for automated abstraction of hardware and software systems. Originally designed for verification of hardware designs, this technique has been successfully utilized for software verification as well. Particularly, the SLAM project [BR01] has gained attention and has demonstrated the effectiveness of software verification for device drivers. The BLAST tool [HJMS02] and the MAGIC tool [CCG+03] have been applied successfully in domains of security protocols and real-time operating-system kernels.

The CEGAR paradigm was introduced in [CGJ+00]. 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, it is abstracted 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: Either the error label is not reachable, then we know that it is not reachable in the concrete program either and the CEGAR process terminates. Or it is reachable in the abstract program, illustrated by means of 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 not spurious (*real* counterexample), it can be reported to the user and the process terminates. If it is spurious, then

* This work was partially supported by the DFG project *Algorithms for Software Model Checking*.

suitable new predicates have to be introduced to refine the abstraction such that this counterexample is excluded in future predicate abstractions.

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 the BDD-based model-checker Moped, a combined reachability and LTL model-checker for symbolic pushdown systems [Sch02].

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), as it is also done by compilers. Thus, we reduce an infinite data space to a finite, but possibly still large data space. Then we use CEGAR to reduce the state space even further. Whereas, in the first step, we might lose some bugs that occur only with large numbers, no precision is lost in the second step, because the abstraction is appropriately refined during the process. Since we do not change the procedural structure in both steps, recursion may always 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 other 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 [BR01], 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. [CCG$^+$03] does not use BDDs at all, but relies on SAT solvers and theorem provers. Also [HJMS02,HJMM04] 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 [BR01].

Another feature of our work is the use of multiple counterexamples in a single refinement step. Moped constructs a “witness graph” (see [RSJM05]) 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 this DAG for abstraction refinement, not only a single counterexample. If the counterexample DAG contains a real (non-spurious) counterexample, it is reported. Otherwise we compute predicates that ensure that none of the counterexamples in the DAG will occur again in future abstractions. In [GKMH$^+$03], multiple counterexamples are also used in a CEGAR scheme, but not for software and not in a DAG structure.

For the predicate generation we use Craig interpolation (see [HJMM04,McM05]). In contrast to [HJMM04], 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 flexibility for heuristics about *which* interpolants to use, since Craig interpolants are, in general, not unique.

Organization of the Paper. This paper proceeds as follows. In Sect. 2 we investigate Craig interpolation for propositional logics and derive computation schemes that are suitable for BDDs. In Sect. 3, symbolic pushdown systems, a model for sequential programs, are reviewed. In Sect. 4, the techniques of Sect. 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 Sect. 6 and conclude in Sect. 7.

2 Craig Interpolation

In [McM03,HJMM04], Craig interpolation was used to automatize abstraction refinement. As in [McM03] (and in contrast to [HJMM04], 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 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 $Occ(I) \subseteq Occ(F) \cap Occ(G)$.*

Craig's Interpolation Theorem [Cra57] states that interpolants always exist, but they are not unique. In [McM03], 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 $I \vee I'$ and $I \wedge I'$ (see also [Kie05]). It follows that "the strongest interpolant" and "the weakest interpolant", as defined below, exist and are unique.

Definition 2. *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 implied by any other interpolant.*

Clearly, $SI(F, G) \models WI(F, G)$ holds. Proposition 1 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 any 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.

Proposition 1 (Strongest and Weakest Interpolants). *Let* (F, G) *be a formula pair with* $F \wedge G$ *unsatisfiable. Then* $SI(F, G) \equiv F \uparrow G$ *and* $WI(F, G) \equiv (\neg G) \downarrow F$*.*

Proof.

- $SI(F, G) \equiv F \uparrow G$:
  We show that $I = F \uparrow G$ is the strongest formula that is implied by F and contains only variables in G. Then, $I \equiv SI(F, G)$, because a strongest interpolant exists.
  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. $\mathcal{A}(I) = \mathbf{true}$. Then, $\mathcal{A}$ can be extended to an assignment $\mathcal{A}^+$ s.t. $\mathcal{A}^+(F) = \mathbf{true}$. Since $F \models J$, we have $\mathcal{A}^+(J) = \mathbf{true}$ and $\mathcal{A}(J) = \mathbf{true}$.
- $WI(F, G) \equiv (\neg G) \downarrow F$: Analogously. □

In the next sections, we consider the following problem: Given a formula $F = F_1 \wedge \ldots \wedge F_n$, determine if F is unsatisfiable, and if so, find interpolants for the pairs (G^i, G_i), $i \in \{1, \ldots, n\}$, where $G^i := F_1 \wedge \ldots \wedge F_i$ and $G_i := F_{i+1} \wedge \ldots \wedge F_n$. We show that strongest and weakest interpolants for (G^i, G_i) can be computed iteratively.

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

(i) F *is unsatisfiable iff* $I_n \equiv \mathbf{false}$ *iff* $J_0 \equiv \mathbf{true}$.
(ii) If F *is unsatisfiable, then* $I_i \equiv SI(G^i, G_i)$ *and* $J_i \equiv WI(G^i, G_i)$.

Proof. We first show by induction that $I_i \equiv G^i \uparrow G_i$. In the base case we have $I_0 \equiv \mathbf{true} \equiv \mathbf{true} \uparrow G_0$. For the induction step, let $i + 1 > 0$. Then we have

$$
\begin{array}{rll}
I_{i+1} & \equiv (I_i \wedge F_{i+1}) \uparrow G_{i+1} & \text{(definition } I_{i+1}) \\
& \equiv ((G^i \uparrow G_i) \wedge F_{i+1}) \uparrow G_{i+1} & \text{(induction hypothesis)} \\
& \equiv ((G^i \wedge F_{i+1}) \uparrow G_i) \uparrow G_{i+1} & \text{(the variables not occurring in } G_i \\
& & \quad \text{do not occur in } F_{i+1}) \\
& \equiv (G^i \wedge F_{i+1}) \uparrow G_{i+1} & (Occ(G_{i+1}) \subseteq Occ(G_i)) \\
& \equiv G^{i+1} \uparrow G_{i+1} & \text{(definition } G^{i+1}) \ .
\end{array}
$$

Now, (i) follows immediately and (ii) is a consequence of Prop. 1. The statements about J_i are proved analogously. □

Now, 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 $SI(G^i, G_i)$ or $WI(G^i, G_i)$.

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

Definition 3 (Tracking Property). *Let $F_1 \wedge \ldots \wedge F_n$ be unsatisfiable, and let K_i be interpolants for (G^i, G_i). We say that the family $\{K_i\}$ satisfies the* tracking property *if $K_i \wedge F_{i+1} \models K_{i+1}$.*

Proposition 3. *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 $(I_i \wedge F_{i+1}, G_{i+1})$,*
*$J_n :=$ **false**, $J_{i-1} :=$ any interpolant for $(G^{i-1}, \neg(F_i \rightarrow J_i))$.*
Then $\{I_i\}$ and $\{J_i\}$ are interpolants for (G^i, G_i) and satisfy the tracking property.

Proof. We prove the statement about $\{I_i\}$ by induction. In the base case we have $I_0 \equiv$ **true**, which is an interpolant for $(\mathbf{true}, G_0)$, because G_0 is unsatisfiable. For the induction step, let $i + 1 > 0$. The tracking property $(I_i \wedge F_{i+1} \models I_{i+1})$ holds, because I_{i+1} is an interpolant for $(I_i \wedge F_{i+1}, G_{i+1})$, which is, by induction hypothesis, an unsatisfiable pair. Thus, we have

$$\begin{aligned} G^{i+1} &\equiv G^i \wedge F_{i+1} & \\ &\models I_i \wedge F_{i+1} & \text{(by induction hypothesis)} \\ &\models I_{i+1} & \text{(by construction of } I_{i+1}\text{)} \\ &\models \neg G_{i+1} & \text{(by construction of } I_{i+1}\text{)} \end{aligned}$$

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

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

Finally, Prop. 4 shows the interplay between interpolants and disjunction:

Proposition 4.

(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. □

2.2 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 4. *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 $OccSem(F)$.*

One could strengthen the notion of interpolants accordingly (by replacing *Occ* by *OccSem* in Def. 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 $OccSem(I) = OccSem(J)$. 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 $OccSem(I) \neq OccSem(J)$, 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 $OccSem(I_n) = OccSem(J_n)$.

Definition 5. *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 $OccSem(\exists Z.F) = OccSem(\forall Z.\neg G)$. We call $OccSem(\exists Z.F)$ a* conciliating set *in this case.*

The algorithm in Fig. 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* */
$I := F$; $J := \neg G$; $Z := OccSem(F) \cup OccSem(G)$
repeat $X := OccSem(I) \setminus OccSem(J)$; $I := \exists X.I$; $Z := Z \setminus X$
$Y := OccSem(J) \setminus OccSem(I)$; $J := \forall Y.J$; $Z := Z \setminus Y$
until $Y = \emptyset$
return (Z, I, J)

Fig. 1. Computation of conciliated interpolants

Given a pair of formulas, the pair of conciliated interpolants is not unique. Proposition 5 characterizes the pair computed by the algorithm.

Proposition 5.

(i) Let C_1, C_2 be the conciliating sets of the pairs I_1, J_1 and I_2, J_2 of conciliated interpolants. 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 Fig. 1 computes the unique maximal conciliating set.

Proof.

(i) "$\Leftarrow$": Clearly, by Def. 5, a pair of conciliated interpolants (I, J) determines their unique conciliated set $\mathit{OccSem}(I) = \mathit{OccSem}(J)$.
"$\Rightarrow$": Let $V = \mathit{OccSem}(F) \cup \mathit{OccSem}(G)$. Let $C \subseteq V$ be a conciliating set for (F, G), i.e., let $C = \mathit{OccSem}(I) = \mathit{OccSem}(J)$ with $I = \exists Z.F$ and $J = \forall Z.G$ for some set $Z \subseteq V$. We show $I \equiv \exists(V \setminus C).F$. (Showing $J \equiv \forall(V \setminus C).G$ 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, i.e., $I \equiv \exists(V \setminus C).I \equiv \exists((V \setminus C) \cup Z).F \equiv \exists(V \setminus C).F$.

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

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

Thus, for Z to be conciliating, it remains to show:

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

Let $v \in V$.
Case 1: $v \in Z$. Then w.l.o.g. $v \in X$. Since X is conciliating, we have $v \in \mathit{OccSem}(\exists(V \setminus X).F) = \mathit{OccSem}(\forall(V \setminus X).G)$,
thus $v \in \mathit{OccSem}(\exists(V \setminus Z).F)$ and $v \in \mathit{OccSem}(\forall(V \setminus Z).G)$.
Case 2: $v \notin Z$.
Then $v \notin \mathit{OccSem}(\exists(V \setminus Z).F)$ and $v \notin \mathit{OccSem}(\forall(V \setminus Z).G)$.

Hence, in both cases we have

$$v \in \mathit{OccSem}(\exists(V \setminus Z).F) \iff v \in \mathit{OccSem}(\forall(V \setminus Z).G).$$

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

(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 gets quantified out s.t. it no longer occurs neither in I nor in J, then it also gets 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. $\square$

One may argue that, since we are interested in simple interpolants, we would like to compute a *minimal* conciliating set. Unfortunately, there may be several. We can compute one by means of a greedy algorithm that tries to quantify out more and more variables. The interpolants produced by such a procedure might be "simpler", but could strongly depend on the arbitrarily chosen variable order.

In the context of abstraction refinement, one can use the algorithm from Fig. 1 as interpolation (and simplification) method when computing a family of interpolants according to Prop. 3. Thus, the tracking property is satisfied.

3 Symbolic Pushdown Systems

As our program model, we use symbolic pushdown systems (SPDSs) [Sch02].

Definition 6 (SPDS[1]). *An SPDS is a quadruple $(G, \mathit{\Gamma}_0 \times L, \mathit{\Delta}, \gamma_0)$, where*

- $G = \{\mathbf{true}, \mathbf{false}\}^{n_G}, n_G \geq 0$, *is the set of global variable valuations,*
- $\mathit{\Gamma}_0$ *is a set of* control points*,*
- $L = \{\mathbf{true}, \mathbf{false}\}^{n_L}, n_L \geq 0$, *is the set of local variable valuations,*
- $\mathit{\Delta}$ *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* $0 \leq n \leq 2, \gamma, \gamma_1, \ldots, \gamma_n \in \mathit{\Gamma}_0$ *and* $R \subseteq (G \times L) \times (G \times L^n)$*,*
- $\gamma_0 \in \mathit{\Gamma}_0$ *is the start address.*

SPDSs model (possibly recursive) programs with procedures. 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. In our setting, they are given as BDDs.

The right side of the rules can consist of zero, one or two control points. Whereas a rule with one control point on the right side describes an intraprocedural statement, a rule with two control points on the right side describes a procedure call, a *push*: γ_1 is the start address of the newly called procedure, and γ_2 the return address of the calling procedure. Parameter passing can be encoded in the relation R of the rule by initializing the local variables of the called procedure. A rule with zero statements is the termination of a procedure, a *pop*. Return values can be encoded in the relation R of the rule by restricting the global variables. SPDSs are discussed in greater detail in [Sch02] and [Kie05].

Example 1. Consider the procedures in Fig. 2. 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 side. The start address is $m0$.

`procedure m`		
`m0:` $L := L \cdot (L+1)$	$\langle m0\rangle \hookrightarrow \langle m1\rangle$	$(L' = L \cdot (L+1) \ \wedge\ G' = G)$
`m1: call f(`L`)`	$\langle m1\rangle \hookrightarrow \langle f0, m2\rangle$	$(L'' = A' = L \ \wedge\ G' = G)$
`m2: if` $G \neq 0$ `then goto error`	$\langle m2\rangle \hookrightarrow \langle \mathit{error}\rangle$	$(G \neq 0 \ \wedge\ G' = G)$
`procedure f(`A`)`	$\langle f0\rangle \hookrightarrow \langle f1\rangle$	$(A \text{ even} \ \wedge\ G' = G)$
`f0: if` A `even then`	$\langle f1\rangle \hookrightarrow \langle f3\rangle$	$(A' = 0 \ \wedge\ G' = G)$
`f1:` $A := 0$	$\langle f0\rangle \hookrightarrow \langle f2\rangle$	$(A \text{ odd} \ \wedge\ G' = G)$
`f2: else` $A := 561$	$\langle f2\rangle \hookrightarrow \langle f3\rangle$	$(A' = 561 \ \wedge\ G' = G)$
`f3:` $G := A$	$\langle f3\rangle \hookrightarrow \langle\rangle$	$(G' = A)$

Fig. 2. Two simple procedures along with an equivalent SPDS

[1] This definition is slightly more restrictive than in [Sch02].

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.

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 [RSJM05] 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 Fig. 3. 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 side of Fig. 3. (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 not. We call the DAG spurious in the first case.

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

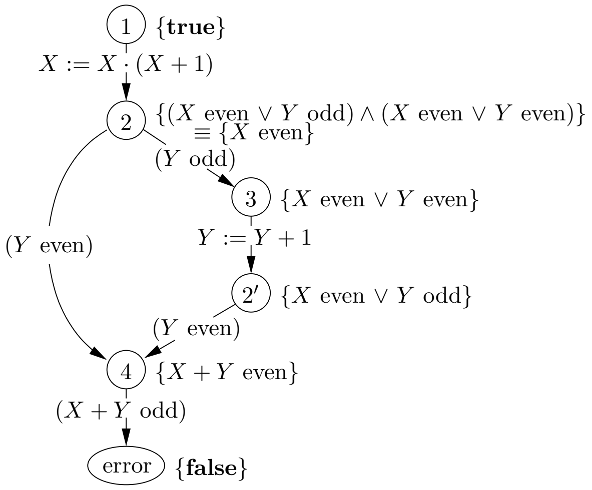

Fig. 3. Program and counterexample DAG with weakest interpolants

Let D be a DAG for the rest of the section. We describe our predicate generation method in three steps: for single counterexamples without procedures, for counterexample DAGs without procedures, and finally for counterexamples 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 in such a way that, for every edge (n_1, R, n_2) in D, $\{P_{n_1}\}\ R\ \{P_{n_2}\}$ is a valid Hoare triple (recall that a SPDS rule R corresponds to a program instruction).
– 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:

$$\begin{array}{l}
\langle 0\rangle \hookrightarrow \langle 1\rangle\ (x' \wedge (y' \leftrightarrow y) \wedge (z' \leftrightarrow z)) \\
\langle 1\rangle \hookrightarrow \langle 2\rangle\ ((x' \leftrightarrow x) \wedge (y' \leftrightarrow x) \wedge (z' \leftrightarrow z)) \\
\langle 2\rangle \hookrightarrow \langle 3\rangle\ ((\neg y \wedge z) \wedge (x' \leftrightarrow x) \wedge (y' \leftrightarrow y) \wedge (z' \leftrightarrow z))
\end{array}$$

```
0: x := true
1: y := x
2: if (¬y ∧ 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:

$$\begin{array}{ll}
F_1 \equiv x_1 \wedge (y_1 \leftrightarrow y_0) \wedge (z_1 \leftrightarrow z_0) & //\ x := \mathbf{true} \\
F_2 \equiv (x_2 \leftrightarrow x_1) \wedge (y_2 \leftrightarrow x_1) \wedge (z_2 \leftrightarrow z_1) & //\ y := x \\
F_3 \equiv (\neg y_2 \wedge z_2) \wedge (x_3 \leftrightarrow x_2) \wedge (y_3 \leftrightarrow y_2) \wedge (z_3 \leftrightarrow z_2) & //\ \mathtt{assume}(\neg y \wedge z)
\end{array}$$

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 Prop. 2 show that F_D is indeed unsatisfiable and yield the following strongest and weakest interpolants:

$$\begin{array}{rll}
I_1 = & SI(G^1, G_1) \equiv \exists\{y_0, z_0, y_1\}.F_1 & \equiv x_1 \\
I_2 = & SI(G^2, G_2) \equiv \exists\{x_1, z_1\}.(SI(G^1, G_1) \wedge F_2) & \equiv (x_2 \wedge y_2) \\
J_2 = & WI(G^2, G_2) \equiv \forall\{x_3, y_3, z_3\}.\neg F_3 & \equiv (y_2 \vee \neg z_2) \\
J_1 = & WI(G^1, G_1) \equiv \forall\{x_2, y_2, z_2\}.(F_2 \rightarrow WI(G^2, G_2)) & \equiv (x_1 \vee \neg z_1)
\end{array}$$

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 (G^n, G_n), which is in fact a predicate over variable values at n. For instance, the interpolants $SI(G^2, G_2)$ and $WI(G^2, G_2)$, or any other interpolant for this pair, can only contain logical variables common to G^2 and G_2, which must necessarily have index 2. These logical variables refer to the values of the program variables after the execution of $x := \mathbf{true}; y := x$ and before the execution of $\texttt{assume}(\neg y \wedge z)$.

Fact 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 $\{\mathbf{true}\}c_1\{P_1\}c_2\{P_2\}\ldots\{P_{k-1}\}c_k\{\mathbf{false}\}$ 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

$$\begin{array}{ll} \{\mathbf{true}\}\ x := \mathbf{true}\quad \{x\}\quad y := x\ \ \{x \wedge y\}\ \texttt{assume}(\neg y \wedge z)\ \{\mathbf{false}\} & (I_i), \\ \{\mathbf{true}\}\ x := \mathbf{true}\ \{x \vee \neg z\}\ y := x\ \{y \vee \neg z\}\ \texttt{assume}(\neg y \wedge z)\ \{\mathbf{false}\} & (J_i). \end{array}$$

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 Prop. 3 guarantee the tracking property and lead to the Hoare annotation $\{\mathbf{true}\}\ x := \mathbf{true}\ \{x\}\ y := x\ \{y\}\ \texttt{assume}(\neg y \wedge z)\ \{\mathbf{false}\}$.

4.2 Multiple Counterexamples

We now extend the techniques from Sect. 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 Fig. 3. The main addition to the technique from Sect. 4.1 is the disjunction at control point 4, where two branches of the DAG merge:

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

As before, D is spurious if and only if F_D is unsatisfiable [Kie05]. 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 its formula pair (F, G). In the example above, P_3 is an interpolant for the formula pair (F_3, G_3), where

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

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, in above example, variable values with index 3.

Strongest and weakest interpolants at each control point in D can be computed in a stepwise way as sketched in Props. 2 and 4.

In the example, the predicates in curly brackets in Fig. 3 are weakest interpolants. Proposition 4 (ii) is used to compute the interpolant at point 2, as sketched in the figure. Since the predicate computed at 1 turns out to be **true**, one can infer (cf. Prop. 2) 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, we have for the whole DAG D the Hoare triple $\{\textbf{true}\}\ D\ \{\textbf{false}\}$, which is an alternative way to state the spuriousness of D.

Example 2. Sometimes, DAGs contain exponentially many counterexamples. Consider, for instance, the program in Fig. 4. This program contains $2^3 = 8$ different paths to the error label, which are all shown in the DAG on the right side.

Using single counterexamples only and with a simple predicate generation strategy (weakest or strongest interpolants), we need in this case $2^3 - 1 = 7$ refinement cycles (essentially one for each single path). This number grows 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 Fig. 4, 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 show how to handle 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 Sect. 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:

```
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 ¬A then error
```

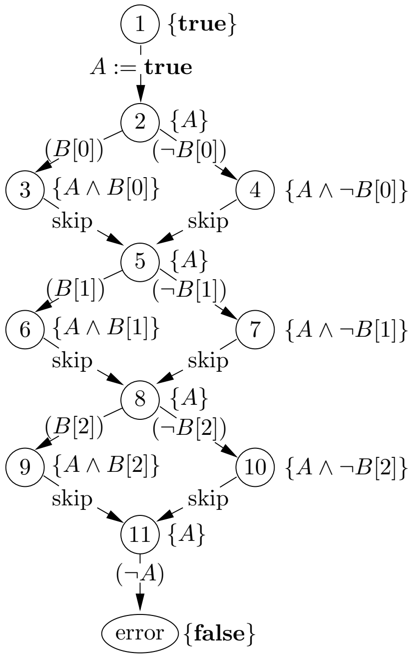

Fig. 4. Program with many paths

– To generate the abstract transition rules in a simple and efficient way (see Sect. 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 goals are to ensure that the predicates P_n at each node n are independent of the calling context and still satisfy the tracking property.

– 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_0 corresponds to the entry point of a procedure, we ensure $(g_{in} \leftrightarrow g) \wedge (l_{in} \leftrightarrow l) \models P_{n_0}$.
– 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 and the saved local data.
– If an edge from node n' is labeled by transition rule $Pop(g, l, g')$ (a return statement) and n is the node at which the corresponding call took place, we first generate an interpolant $P_{<n}(g_{in}, l_{in}, g')$, effectively a predicate that argues about the effect of the callee, s.t. $P_{n'} \wedge Pop \models P_{<n}$. Then, if n'' is the target node of the edge, we ensure that $P_{>n} \wedge P_{<n} \models P_{n''}$.
– 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.

Figure 5 gives an example for a (spurious) counterexample DAG to the SPDS in Fig. 2, 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.

To explain how the predicates are computed, we consider a generic procedure call/return pair, see Fig. 6. A procedure `m` calls a procedure `f` and is resumed after the termination of `f`. For simplicity of our description how interpolants are computed, 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.

Whereas the top-down computation of strongest interpolants (left-hand side of Table 1) is quite straightforward and resembles Prop. 2, the bottom-up weakest interpolants computation scheme (right-hand side of Table 1) needs 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. We stated earlier that we want to guarantee $W_{>m_i} \wedge P_{<m_i} \models W_{m_{i+1}}$, no matter what the predicate $P_{<m_i}$ will

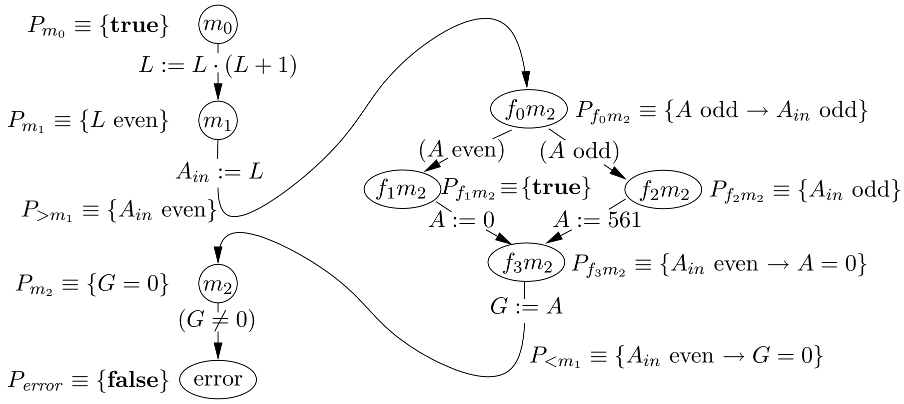

Fig. 5. An example for a counterexample DAG with procedure call

be. The weakest formula over $(g_{m_{in}}, l_{m_{in}}, g_{f_{in}}, l_{f_{in}}, l_{m_{i+1}})$ that satisfies this implication is

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

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

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

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

as stated in Table 1.

On a more intuitive level, we use the most precise precise description of `f`'s effect available to us, namely the strongest interpolant $S_{<m_i}$, 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 $S_{<m_i}$ to $W_{<m_i}$ by overapproximating `f`'s effect in terms of the

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	$\forall\{g_{f_{in}}, l_{f_{in}}, l_{m_{i+1}}\}.(Push \rightarrow W_{>m_i})$
$S_{>m_i}/W_{>m_i}$	$\exists\{g_{m_i}, l_{m_i}\}.(S_{m_i} \wedge Push)$	$\forall\{g_{m_{i+1}}\}.(S_{<m_i} \rightarrow W_{m_{i+1}})$
$S_{f_0m_i}/W_{f_0m_i}$	$(g_{f_{in}} \leftrightarrow g_{f_0}) \wedge (l_{f_{in}} \leftrightarrow l_{f_0})$	$\forall\{g_{f_1}, l_{f_1}\}.(R_{f_1} \rightarrow W_{f_1})$
$S_{f_1m_i}/W_{f_1m_i}$	$\exists\{g_{f_0}, l_{f_0}\}.(S_{f_0m_i} \wedge R_{f_1})$	bottom-up
$S_{f_jm_i}/W_{f_jm_i}$	top-down	$\forall\{g_{m_{i+1}}\}.(Pop \rightarrow W_{<m_i})$
$S_{<m_i}/W_{<m_i}$	$\exists\{g_{f_j}, l_{f_j}\}.(S_{f_jm_i} \wedge Pop)$	$\forall\{g_{m_{in}}, l_{m_{in}}, l_{m_{i+1}}\}.(W_{>m_i} \rightarrow W_{m_{i+1}})$
$S_{m_{i+1}}/W_{m_{i+1}}$	$\exists\{g_{f_{in}}, l_{f_{in}}\}.(S_{>m_i} \wedge S_{<m_i})$	bottom-up
S_{m_n}/W_{m_n}	top-down	**false**

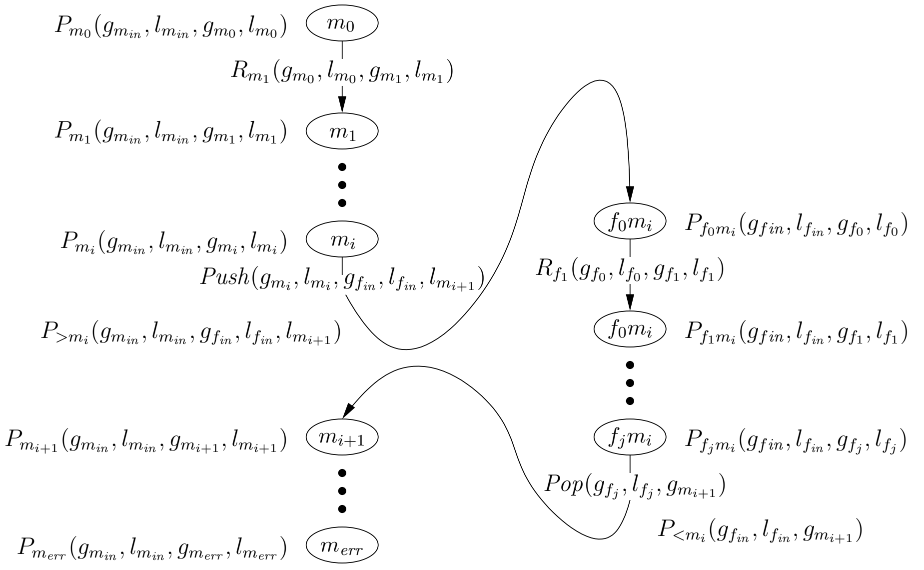

Fig. 6. A counterexample DAG with procedure call

given counterexample DAG: We need not say more about $W_{<m_i}$ than that it transforms $W_{>m_i}$ to $W_{m_{i+1}}$.

In [Kie05], we also discuss how to handle counterexample DAGs that contain calls of procedures that are not finished. Such a structure is simpler than the previously discussed one and can be easily accommodated.

Our implementation improves the efficiency of the predicate generation by working *directly* on the witness graphs discussed in [RSJM05]. This data structure represents counterexample DAGs in a compact way by reusing nodes from procedures that are called multiple times. For an example, consider the program in Fig. 7 (simplified from [Sch02]), whose first abstraction contains an exponentially (in n) long spurious counterexample.

In [Sch02] and [RSJM05] 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, as described in [Kie05], which leads to a linear number of predicates in the above example.

We conclude that the witness graph structure gives us two *different* exponential savings. In Sect. 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.

```
bool G;

procedure main {
    G := true
    level₁()
    level₁()
    if (¬G) then
        error
}
```

```
procedure level₁() {
    level₂()
    level₂()
}
⋮
procedure levelₙ₋₁() {
    levelₙ()
    levelₙ()
}
procedure levelₙ() {
    G := ¬G
}
```

Fig. 7. Program whose first abstraction includes an exponentially long spurious counterexample

5 Computing the Abstract SPDS

In each CEGAR cycle, we derive predicates to refine our abstraction. By the methods of Sect. 4, each predicate naturally belongs to a control point. Thus, as in [HJMM04], we maintain for each control point a list of predicates that are useful there. In this section we sketch how to compute an (overapproximating) abstract SPDS given a concrete one along with the predicate lists.

Consider the example SPDS of Sect. 4.1. 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 $[2] := l_2 \leftrightarrow y$. A *concretization* $[i]$ would be a conjunction of more than one equivalence if the predicate list of control point i consists of more than one predicate.

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

$$\exists\{x,y,z,x',y',z'\}.\big((l_2 \leftrightarrow x) \wedge ((x' \leftrightarrow x) \wedge (y' \leftrightarrow x) \wedge (z' \leftrightarrow z)) \wedge (l_3' \leftrightarrow y')\big) \equiv l_2' \leftrightarrow l_1.$$

We can save variables, 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:

$$\langle 0 \rangle \hookrightarrow \langle 1 \rangle\ (l')$$
$$\langle 1 \rangle \hookrightarrow \langle 2 \rangle\ (l' \leftrightarrow l)$$
$$\langle 2 \rangle \hookrightarrow \langle 3 \rangle\ (\neg l)$$

```
0: l := true
1: skip
2: if ¬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:

{**true**} x := **true** {x} y := x {y} `assume`($\neg y \wedge z$) {**false**} translates into {**true**} l := **true** {l} `skip` {l} `assume`($\neg l$) {**false**}. 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.

The procedural case is more involved and is omitted for space reasons. We sketch only one important concept here: The effect of a called procedure (the predicate $P_{<n}$) must be captured in an abstract variable and inspected by the caller in order to incorporate the procedure effect into its local abstract variable values. The details can be found in [Kie05].

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 [SSE05]. We did not compare our program with existing CEGAR tools, since the assumptions of tools like BLAST and SLAM (infinite variable ranges, theorem provers) differ significantly from ours (finite variable ranges).

6.1 Locking Example

Figure 8 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. We want to model-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 8 bits of bit width).

```
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);
}
```

Fig. 8. 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 abstraction	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 abstraction 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 grows 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 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 [Wei98] 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. (This is not a bug though.) Using 4 bits of bit width and 64 bits for Moped's heap representation, Moped without abstraction needs 143 seconds to find an error trace, whereas with CEGAR, only 7.4 seconds are needed (memory consumption: about 2.5 Mio. BDD nodes in both cases). A refinement is not necessary. Moped's performance without abstraction quickly degrades with growing heap size, whereas with abstraction, the influence of the heap size is small.

We also discovered cases where with our predicate generation heuristics, abstraction did not pay off, particularly if complicated properties are checked.

7 Conclusions

Whereas Craig interpolation has been used for CEGAR in SAT solver and theorem prover contexts, we found that it is useful as well to enhance a BDD-based model-checker. Strongest and weakest interpolants, which are defined independently from other tools, form a frame 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, to be able 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 [HJMS02,Kie05] and the incorporation of dataflow information to detect relevant counterexample parts [JM05].

Acknowledgement. We thank Dejvuth Suwimonteerabuth for his great support with jMoped.

References

[BR01] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01*, LNCS 2057, pages 103–122, 2001.

[CCG+03] 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.

[CGJ+00] 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.

[Cra57] W. Craig. Linear reasoning. A new form of the Herbrand-Genzen theorem. *Journal of Symbolic Logic*, 22:250–268, 1957.

[GKMH+03] 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.

[HJMM04] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proc. POPL'04*, pages 232–244. ACM Press, 2004.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL'02*, pages 58–70. ACM Press, 2002.

[JM05] R. Jhala and R. Majumdar. Path slicing. In *Proc. of PLDI '05*, pages 38–47. ACM, 2005.

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

[McM03] K.L. McMillan. Interpolation and SAT-based Model Checking. In *Proc. CAV'03*, LNCS 2725, pages 1–13. Springer, 2003.

[McM05] K.L. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of TACAS 2005*, LNCS 3440, pages 1–12. Springer, 2005.

[RSJM05] 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.

[Sch02] S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, TU München, 2002.

[SSE05] 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.

[Wei98] M.A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison-W., 1998.