Using Statically Computed Invariants inside the Predicate Abstraction and Refinement Loop

Himanshu Jain[1,2], Franjo Ivančić[1], Aarti Gupta[1], Ilya Shlyakhter[1], and Chao Wang[1]

[1] NEC Laboratories America, 4 Independence Way, Suite 200, Princeton, NJ 08540
[2] Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213

Abstract. Predicate abstraction is a powerful technique for extracting finite-state models from often complex source code. This paper reports on the usage of statically computed invariants inside the predicate abstraction and refinement loop. The main idea is to *selectively strengthen* (conjoin) the concrete transition relation at a given program location by efficiently computed invariants that hold at that program location. We experimentally demonstrate the usefulness of transition relation strengthening in the predicate abstraction and refinement loop. We use invariants of the form $\pm x \pm y \leq c$ where c is a constant and x, y are program variables. These invariants can be discovered efficiently at each program location using the octagon abstract domain. We observe that the abstract models produced by predicate abstraction of strengthened transition relation are more precise leading to fewer spurious counterexamples, thus, decreasing the total number of abstraction refinement iterations. Furthermore, the length of relevant fragments of spurious traces needing refinement shortens. This leads to an addition of fewer predicates for refinement. We found a consistent reduction in the total number of predicates, maximum number of predicates tracked at a given program location, and the overall verification time.

1 Introduction

Predicate abstraction [13] is a powerful technique for extracting finite-state models from often complex source code. It abstracts data by keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. In most predicate abstraction and refinement based tools [4, 14, 6, 17], spurious behavior in the abstract model is removed by adding new predicates or making the relationships between existing predicates more precise. Thus, even the information that can be discovered efficiently using other abstract domains (e.g., numerical abstract domains [10, 22]) is learned only through multiple refinement iterations in the form of new predicates.

A large number of predicates poses a problem as both the predicate abstraction computation and the model checking of the abstraction are exponential in the number of predicates. In the SLAM [4] toolkit, this problem is handled by generating coarse abstractions using techniques such as *Cartesian approximation* and the *maximum cube length approximation*. These techniques limit the number of predicates in each theorem prover query. The refinement of the abstraction is carried out by adding new predicates.

If no new predicates are found, the spurious behavior is due to inexact predicate relationships. Such spurious behavior is removed by making the relationships between existing predicates more precise.

The BLAST toolkit [14] introduced the notion of *lazy abstraction*, where the abstraction refinement is completely demand-driven to remove spurious behaviors. When refining an infeasible (spurious) sequence of program statements, BLAST adds new predicates only to basic blocks occurring in the infeasible trace [15]. We refer to this as *localization of predicates*. While BLAST makes use of interpolation, localization of predicates can also be carried out using weakest pre-conditions [17]. On average the number of predicates tracked at each program location is small and thus, the localization of predicates enables predicate abstraction to scale to larger programs.

The techniques described above employ over-approximations of the most precise abstract models to ensure scalability of the individual steps in the abstraction refinement loop. However, over-approximations introduce more spurious counterexamples resulting in an increase in the number of refinement iterations. Even though the refinement process is completely automatic, a large number of refinement iterations can make the entire predicate abstraction and refinement loop inefficient, and often intractable.

This paper makes the following contributions:

- Our main idea is to *strengthen* the *concrete transition relation* at a given program location l using invariants that hold at l. In standard predicate abstraction approaches (not using invariants) each program location is abstracted in isolation, that is, no relationships are assumed between the variables read at that location. Strengthening of the concrete transition relation using invariants provides additional relationships between the variables read at a program location. Thus, the abstract model produced using the strengthened transition relation can be more precise leading to fewer spurious counterexamples as compared to standard approaches.
- We show the efficacy of the above idea by incorporating an abstract domain, namely the *octagon abstract domain* [21, 22], into the predicate abstraction and refinement loop. Octagonal invariants are invariants of the form $\pm x \pm y \leq c$, where x and y are numerical program variables and c is a numerical constant. These invariants can be computed efficiently by the octagon abstract domain. The octagon abstract domain has been used within Astrée [11], and was shown instrumental in reducing the number of false alarms when detecting runtime errors in critical embedded software [22]. The following ideas are needed to make strengthening using octagonal invariants beneficial in practice.
- *Invariant Generation:* Tracking octagonal relationships between a large number of program variables is expensive. In Astrée, the set of program variables is *clustered* into various sets of related variables known as *octagon packs*. The octagonal relationships between all octagon pack variables are computed separately for each octagon pack. The size of each octagon pack is kept small, so that the computation of octagonal relationships between the variables of an octagon pack does not become a bottleneck. We describe a new clustering strategy which attempts to create octagon packs containing program variables which may likely appear in predicates and their weakest pre-conditions through abstraction refinement.

- *Invariant Selection:* After invariant generation there can be many octagonal relationships that hold at each program location. Using all invariants that hold at program location l to strengthen the transition relation at program location l may not be beneficial. This is because providing too many additional relationships in form of invariants can potentially increase the burden on the decision procedure used for abstraction computation and simulation of abstract counterexamples. We describe a heuristic for *selecting* the invariants that are used for strengthening the transition relation at a given program location.

Further related work: The idea of using statically computed invariants during abstraction has been mentioned before [5, 9, 23]. Both Bensalem et al. [5] and Saïdi [23] note that using invariants during abstraction can produce abstract models with fewer transitions and less reachable states. However, in [5, 9] the invariants to be used during abstraction need to be supplied by the user. An invariant generation technique is proposed in [23] which produces quantified invariants at each program location. However, the tradeoffs involved in efficiently using the computed invariants in the abstraction refinement loop are not discussed.

Constraints of the form $\pm x \pm y \leq c$ arise frequently in software verification. Seshia et al. [24] observe that most of the linear arithmetic constraints arising in software verification have the form $x - y < c$. Ball et al. [3] report that most of the queries that arise during the refinement process of SLAM are of the form $\pm x \pm y \leq c$. However, to the best of our knowledge none of the predicate abstraction and refinement tools for C code [4, 14, 6, 17] use (octagonal) invariants during verification. Fischer et al. [12] describe a technique for obtaining a path sensitive version of any data flow analysis by using predicated lattices. Instead, we use transition relation strengthening as a means of incorporating information from other data flow analysis into the predicate abstraction and refinement loop.

2 Motivating example

We use the counterexample-guided abstraction and refinement loop [19, 7, 4] to check safety properties (such as unreachability of error labels) in C programs. Consider the C program shown in Fig. 1(a) with variables x, y, z considered as integers. Assume that the statements not shown do not affect the variables x, y, z. Predicate abstraction of the C program with respect to an empty set of predicates is shown in Fig. 1(b). Observe that the control flow in both the abstract model and the C program is the same. Since the initial set of predicates is empty we cannot track the value of the conditions at program locations 1 and 10 in the abstract model precisely. Thus, the conditions at program locations 1 and 10 in the C program are replaced by non-deterministic choice (represented as * in the figure) in the abstract model. All assignments in the C program are replaced by `skip` statements in the abstract model. A skip statement at a program location l in the abstract model means that the statement at program location l in the C program has no effect on the predicates being tracked in the abstract model. The `ERROR` label in the C program is preserved in the abstract model.

Model checking of the abstraction in Fig. 1(b) produces an *abstract counterexample* which goes through all program locations starting from 1 to 11 (`ERROR`). Since the abstract counterexample may or may not correspond to a real bug in the C program, it is

```
PC
1: if (x > y) {
2:   y = y + 1;
...
5:   z = y;
...
10:  if (x < z)
11:    ERROR:;
12: }
```

(a)

```
PC
1: if (*) {
2:   skip;
...
5:   skip;
...
10:  if (*)
11:    ERROR:;
12: }
```

(b)

```
PC
1: assume (x > y);
2: y = y + 1;
...
5: z = y;
...
10: assume (x < z);
```

(c)

```
PC Invariants
1:
2:  x>y
..  x>y-1
5:  x>y-1
..  x>y-1, z=y, x>z-1
10: x>y-1, z=y, x>z-1
11:
12:
```

(d)

```
PC
1: if (*) {
2:   skip;
...
5:   skip;
...
10:  if (b)     [(PC = 10) → ¬b]
11:    ERROR:;
12: }
```

(e)

Fig. 1. PC stands for program counter. (a) C program. (b) Abstraction of C program with respect to an empty set of predicates. (c) Infeasible program trace corresponding to abstract counterexample in (b). (d) The computed invariants at every program location. (e) Refined abstraction with the use of invariants. This abstract model has no path to the `ERROR` label.

checked if there is a *feasible* sequence of statements in the original C program leading to the `ERROR` label and having the same control flow as the abstract counterexample. The feasibility check is carried out using a decision procedure. For the abstract counterexample produced by model checking the abstraction in Fig. 1(b), the corresponding sequence of statements in the C program is shown in Fig. 1(c). The `assume` statement shows which branch of the `if` statement was taken in the abstract counterexample.

Consider the program trace shown in Fig. 1(c). The relationship $x > y$ holds at the program location 2 (before `y=y+1` is executed). Variable y is incremented at program location 2, thus, $x > y - 1$ holds after program location 2 (after `y=y+1`). Variable z is assigned y at location 5, so $x > z - 1$ holds after program location 5. Since x, y, z are integers, we have $x \geq z$ after program location 5. The relationship $x \geq z$ contradicts with the `assume` statement at location 10 ($x < z$). Thus, the trace in Fig. 1(c) is an infeasible trace. In order to eliminate the infeasible trace shown in Fig. 1(c) the refined abstract model needs to track the value of the condition $x < z$ at program location 10 precisely, as it guards the `ERROR` label. This is done by introducing new predicates in most tools.

Using the technique described in [15, 17] the infeasible trace shown in Fig. 1(c) can be removed by tracking exactly one predicate at each program location from 1 to 10. The technique of [17] will track the following relationships in the abstract model: $x < y + 1$ is false at program location 2 (before `y=y+1`), $x < y$ is false from location 3 till 5, $x < z$ is false from location 6 to location 10. Note that even though three new predicates ($x < z, x < y, x < y + 1$) are introduced only the value of one predicate needs to be tracked at each program location. The drawback of these techniques is that predicate relationships need to be tracked for the entire infeasible trace, even at the program locations (3,4,6,7,8,9) not directly involved in the infeasibility of the program trace.

Next we show how the use of efficiently computable invariants (such as *octagonal invariants*) can improve the above techniques. The two variable invariants that hold at various program locations of the program in Fig. 1(a) are shown as annotations in Fig. 1(d). For example, at the program location 10 the relationships $x > y-1, x > z-1, y = z$ hold. The invariants shown can be written as conjunctions of octagonal invariants and can be computed using the octagon abstract domain [21, 22]. For example, $x > y-1$ can be written as $-x+y \leq 2$, and $y = z$ is equivalent to a conjunction of two octagonal invariants $y-z \leq 0$ and $-y+z \leq 0$. The advantages of using the invariants in the predicate abstraction and refinement loop are given below.

- *Reduction in the length of infeasible trace fragments needing refinement:* Let us consider the use of invariants during the detection of infeasible traces. Consider the program trace in Fig. 1(c). Without the use of invariants the trace is infeasible due to statements at location 1, 2, 5, 10. The refinement procedure generates new predicates by looking at all four statements. However, with the aid of invariants the statement at location 10 is itself infeasible because the invariant $x > z-1$ holds at location 10 (see Fig. 1(d)). Thus, the refinement procedure only needs to look at a fragment of the trace consisting of only the statement at program location 10.
- *Reduction in the number of predicates needed for refinement:* Without the use of invariants, the refinement schemes of [15, 17] track the value of at least one predicate at each program location from 1 to 10. Using invariants the refinement procedure only looks at program location 10 (PC=10) and the invariants that hold at that location. The condition $x < z$ of the `assume` statement at location 10 of the infeasible trace is introduced as a predicate and its value is tracked only at PC=10 in the refined abstract model shown in Fig. 1(e). The Boolean variable b represents the predicate $x < z$ in the abstract model. The constraint $\neg b$ holds at PC=10 as the invariant $x > z-1$ holds at PC=10 in C program. With the aid of the constraint $(PC = 10) \rightarrow \neg b$ the abstract model of Fig. 1(e) has no path to the `ERROR` label.

Octagon abstract domain alone is precise enough to show that `ERROR` label is unreachable in Fig. 1(a). However, this is not always the case. If the condition at PC=10 in Fig. 1(a) is $2x < z+y$ (not in octagonal form), then the octagon abstract domain cannot show that `ERROR` label is unreachable. Predicate abstraction and refinement loop can still use the octagonal invariants and show the unreachability of `ERROR` label using the abstract model shown in Fig. 1(e), with b representing the predicate $2x < z+y$.

One reason to combine invariants with predicate abstraction, especially in the context of weakest pre-condition based refinement as in [6, 17], is the problem of handling loops efficiently. Often, these techniques model multiple loop unwindings through the use of several related predicates that correspond to different loop unwindings. Instead, certain classes of loop invariants can be computed efficiently [11], and their usage inside the abstraction refinement loop can lead to quicker convergence in presence of loops.

Example: In the C code below we wish to verify the `assert` statement. The use of the loop invariant $x = y$ in the abstraction refinement loop can eliminate the need of numerous predicates of the form $x = 200, y = 200, \ldots, x = 0, y = 0$ which arise when using the weakest pre-condition based refinement. The invariant $x = y$ can be discovered using the octagon abstract domain.

```
1. int x = 200, y = 200;
2. while (x !=0) { x = x - 1; y = y - 1; }
3. assert (y==0);
```

In the above example, interpolant based refinement [15] may or may not succeed in finding $x = y$ as a predicate, due to its dependence on a proof of unsatisfiability of the infeasible trace. This problem is addressed in [18] where a specialized split prover is used to restrict the language of interpolants to avoid divergence and provide a (relatively) complete method for finding predicates. However, the impact of such restrictions and the practical efficiency of a split solver on large examples are not addressed.

3 Transition Relation Strengthening

We operate on a control flow graph of the given program, after various pre-processing steps performed by the F-SOFT tool [16]. Let b denote a basic block in the control flow graph. It can contain multiple assignments or an `assume` statement describing which branch of a condition is taken. Let $T_b(V,V')$ denote the transition relation of basic block b, where V,V' denote the state of program variables before and after executing b, respectively. An *invariant* I_b at basic block b is a Boolean formula over V. Invariant I_b evaluates to true whenever the program counter is at b in any execution of the program. Suppose we have pre-computed a particular set of invariants at each basic block. Let $CI_b(V)$ denote the conjunction of various invariants that hold at basic block b. The idea of *transition relation strengthening* is to use $CI_b(V) \wedge T_b(V,V')$ instead of $T_b(V,V')$ when analyzing b. We refer to $CI_b(V) \wedge T_b(V,V')$ as the *strengthened transition relation* of basic block b and denote it by $ST_b(V,V')$. Invariants over V' are not needed for strengthening the transition relation of b as they are implied by $ST_b(V,V')$. The strengthened transition relation $ST_b(V,V')$ can be used inside the predicate abstraction and refinement loop by using $ST_b(V,V')$ in place of $T_b(V,V')$. We describe this process in more detail below.

Predicate abstraction computation: In predicate abstraction, the variables of the concrete program are replaced by Boolean variables that correspond to a predicate on the variables in the concrete program. These predicates are functions that map a concrete state $V \in S$ into a Boolean value, where S denotes the set of program states. Let $P = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates over the program variables. When applying all predicates to a specific concrete state, one obtains a vector of Boolean values, which represents an abstract state W. We denote this function by $\alpha(V)$. It maps each concrete state into an abstract state and is called an *abstraction function*.

The predicate abstraction of a basic block b is carried out using existential abstraction, i.e., the abstract model can make a transition from an abstract state W to W' iff there is a transition from V to V' after executing basic block b and V is abstracted to W and V' is abstracted to W'. We denote the abstract transition relation obtained by predicate abstraction of basic block b with respect to predicates in P as $\hat{T}_b(W,W')$.

$$\hat{T}_b := \{(W,W') \,|\, \exists V,V' \in S: \ (\alpha(V) = W) \ \wedge \ T_b(V,V') \ \wedge (\alpha(V') = W')\} \qquad (1)$$

Note that the above equation computes the abstraction of b with respect to predicates in P in *isolation*. The term isolation means that no relationships are assumed between the variables in V during abstraction. However, certain relationships may hold between the variables in V when the program execution reaches b. In current predicate abstraction tools, such relationships will be discovered on-demand through multiple refinement iterations, in the form of new predicate relationships in the abstract model. Many of these relationships can however be computed efficiently in the form of invariants. The aim of strengthening is to provide such relationships in the concrete program itself, rather than discovering them in form of predicate relationships in the abstract model. Let $S\hat{T}_b(W,W')$ denote the abstract transition relation obtained by using the strengthened transition relation for basic block b, that is, replacing $T_b(V,V')$ by $ST_b(V,V')$ in Equation 1. The following claim states that predicate abstraction using the strengthened transition relation for b can be more precise than predicate abstraction of b in isolation.

Claim. $\forall b : S\hat{T}_b(W,W') \subseteq \hat{T}_b(W,W')$

The above claim follows from the definition of strengthened transition relation and Equation 1. Consider a concrete program C. Using the strengthened transition relation for each basic block in C during verification does not add any new behaviors to C or remove any existing behaviors from C. This is because strengthening provides invariants which are implicit in C. Let $\hat{C}$ denote the predicate abstraction of C obtained by using $S\hat{T}_b(W,W')$ for every basic block b in C. The following claim then states the soundness of predicate abstraction obtained using the strengthened transition relation.

Claim. Abstraction soundness: $\hat{C}$ is a conservative over-approximation of C.

Simulation of program traces: If the property is violated in the abstract model, we obtain an abstract counterexample from the model checker. In order to check if an abstract counterexample corresponds to a concrete counterexample, a *simulation* step is performed. By ensuring that the control flow in the concrete program is preserved in the abstract model, an abstract counterexample can be mapped back to a sequence Tr of basic blocks $b_1,\ldots,b_k$ in the concrete program, where b_1 is the entry block and b_k contains the `ERROR` label in the given program. Let V_i, V_{i+1} denote the state of program variables before and after executing the basic block b_i, respectively. We say Tr is *feasible* iff there is a real execution of the concrete program which follows the same sequence of basic blocks as Tr. The simulation step checks the feasibility of Tr by checking the satisfiability of the following equation:

$$Sim(Tr) := T_{b_1}(V_1,V_2) \wedge T_{b_2}(V_2,V_3) \wedge \ \ldots \ \wedge T_{b_k}(V_k,V_{k+1}) \tag{2}$$

Claim. The trace Tr is feasible iff $Sim(Tr)$ is satisfiable.

Let $STsim(Tr)$ denote the simulation equation when the strengthened transition relation is used.

$$STsim(Tr) := ST_{b_1}(V_1,V_2) \wedge ST_{b_2}(V_2,V_3) \wedge \ \ldots \ \wedge ST_{b_k}(V_k,V_{k+1}) \tag{3}$$

The following claim states that using the strengthened transition relation for simulation of abstract counterexamples is sound. That is, if Tr is a real counterexample (feasible), then $STsim(Tr)$ is satisfiable, and if Tr is infeasible, then $STsim(Tr)$ is unsatisfiable.

Claim. Simulation soundness: Tr is feasible iff $STsim(Tr)$ is satisfiable.

Let Tr be an infeasible trace when no invariants are used, then Tr is also infeasible when the strengthened transition relation is used (above claim). However, with strengthening it is possible that a sub-sequence Tr' of Tr is itself infeasible. In this case the refinement can be done by looking at only Tr' and the invariants that hold along Tr'. In Section 2 we presented an example where the length of infeasible trace is reduced from 10 to 1 by using the strengthened transition relation. This in turn allows refinement with fewer predicates per program location.

4 Invariants for Transition Relation Strengthening

The octagon abstract domain [21, 22] allows the representation and manipulation of *octagonal invariants*, which have the form $\pm x \pm y \leq c$, where x, y are numerical variables and c is a numerical constant. The octagon abstract domain allows the representation of octagonal relationships between n program variables with $O(n^2)$ memory cost. In order to compute octagonal relationships various *abstract* operators (transfer functions) are needed. The octagon abstract domain provides all the required operators with worst case $O(n^3)$ time cost. We selected octagonal invariants for transition relation strengthening because they can be computed efficiently and are expressive enough to capture many commonly occurring variable relationships [24, 3] and simple loop invariants, important for checking standard properties such as array bounds violation [21]. However, strengthening can also be carried out using other more expressive classes of invariants. Issues involved in the generation and usage of octagonal invariants are discussed below.

4.1 Octagon Packing for Invariant Generation

Computing octagonal relationships between n variables has $O(n^2)$ memory cost per program location and $O(n^3)$ time cost per transfer function. This can become prohibitive when n is large. In Astrée [11] the set of program variables is clustered into various sets of related variables, known as *octagon packs*. The octagonal relationships are computed separately for each octagon pack. The size of each octagon pack is kept small so that the computation of octagonal relationships between the variables in an octagon pack is fast. Octagon packing trades off accuracy of generated invariants for speed, and thus, choosing a right packing strategy is important for the generated invariants to be useful. We have experimented (Section 5.2) with the following octagon packing techniques.

- *Basic block based packing:* We implemented the octagon packing technique used in Astrée as described in [22] (Chapter 8). An octagon pack is associated with each basic block of the control flow graph. All the variables occurring in a basic block (excluding non-linear terms) are made a part of the octagon pack associated with the basic block. If the basic block is a part of a `while`, or `if-then-else` structure, then the variables appearing in the condition of the `while` or `if-then-else` structure are made a part of the octagon pack.
- *Control flow based packing:* We propose a new packing technique that associates an octagon pack with each condition in the control flow graph. Let $oct(c)$ denote the octagon pack corresponding to a condition c at program location l. All numerical variables occurring in c are made a part of $oct(c)$. Then a backward traversal of

the control flow graph is done starting from l. Whenever any variable in $oct(c)$ is updated through an assignment, the variables appearing in the assigned expression are added to $oct(c)$. Thus, the variables in $oct(c)$ affect the value of condition c either directly or indirectly. In the above packing techniques a user specified bound can be used to control the size of an octagon pack.

4.2 Invariant Selection for Strengthening

In general the expectation is that adding invariants would provide a performance improvement for the abstraction computation due to additional pruning of the search space. However, for the same pruning power, a smaller number of invariants is better since that would burden the decision procedure less. On the other hand, the invariants are redundant when we are checking the feasbility of an abstract counterexample. But using invariants can still speed up the feasibility check by providing facts that will otherwise need to be derived by the decision procedure. Using invariants also helps in obtaining smaller infeasible traces for refinement. Therefore, our heuristic is to use fewer invariants so that we get benefit from additional/quicker pruning, without incurring too much overhead due to additional constraints in the decision procedure calls.

For each octagon pack the relationships between the variables appearing in it are tracked at every basic block. This can result in a large number of invariants at every basic block. We apply a heuristic to filter out invariants that are not deemed important for checking the given property. Let I be an invariant that holds at the entry to a basic block b. Let $needed(b,E)$ denote the set of variables whose values need to be tracked at basic block b for checking the reachability of a given error label E. We compute $needed(b,E)$ at each basic block b by performing a syntactic cone-of-influence computation starting from E. We use the following heuristic for selecting the invariants:
InvSelect: Use I to strengthen the basic block b only if all variables appearing in I are present in $needed(b,E)$.

5 Experimental Results

We have implemented these techniques in NEC's F-SOFT [16] verification tool. F-SOFT allows checking the C code for user specified (assert statements) or standard properties (array bound violations, NULL pointer dereferences, use of uninitialized variables). Details about the software modelling in F-SOFT can be found in [16]. We used a 2.8GHz dual-processor Linux machine with 4GB of memory for experiments. Before the abstraction refinement loop starts, we pre-compute the octagonal relationships using the octagon abstract domain library [2]. We use a SAT solver for computing the predicate abstraction [20, 8] and simulation of counterexamples. We report results on TCAS and internal benchmarks. TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system. We used an ANSI-C version of a TCAS component available from Georgia Tech. Even though the preprocessed program has only 224 reachable basic blocks, the number of predicates needed to verify the properties is non-trivial for both F-SOFT and BLAST [1]. We checked 10 different safety properties of the TCAS system using predicate abstraction. None of these properties can be verified by using the octagonal invariants alone. We also analyzed 45 internal industrial benchmarks `SW-1, ..., SW-45` for standard property violations. Some of these benchmarks have more than 1000 reachable basic blocks.

Bench	Default							Strengthen							BLAST		
-mark	Time	Abs	MC	SR	Preds	Cex	I	Time	Abs	MC	SR	Preds	Cex	I	Time	Preds	I
`tcas1a`	87	19	40	28	93/31	11	38	**51**	15	12	24	65/21	7.4	28	102	81/24	35
`tcas1b`	386	49	266	71	137/56	20	54	333	58	177	98	126/49	16	50	**278**	108/36	69
`tcas2a`	87	18	41	30	94/36	11.3	38	**48**	15	11	22	57/18	7.1	26	112	97/29	38
`tcas2b`	**95**	20	41	34	99/34	13.1	39	100	26	27	47	78/27	11.6	37	177	106/31	52
`tcas3a`	164	25	96	43	113/48	13.4	40	**131**	27	51	53	89/31	11.4	36	217	130/37	57
`tcas3b`	**56**	11	26	19	82/27	9.9	28	69	18	19	32	64/21	8.9	28	92	99/26	33
`tcas4a`	334	51	199	84	122/45	14.7	40	**167**	33	70	64	97/33	13	40	515	158/48	104
`tcas4b`	130	27	54	49	88/28	11.2	32	**90**	25	24	41	77/22	10.6	32	303	127/36	47
`tcas5a`	113	26	40	47	96/28	10.3	32	**27**	9	6	12	46/12	6.6	17	100	87/21	29
`tcas5b`	149	29	69	51	98/29	10.4	30	**87**	23	27	37	75/22	9.2	25	139	102/27	39

Table 1. Comparison between three implementations of predicate abstraction and refinement loop. 1) Default: uses the localization of predicates [17]. 2) Strengthen: Uses the strengthened transition relation in the same framework as [17]. 3) BLAST: Results of running BLAST with Craig interpolation options. All times are reported in seconds. ”Abs”, ”MC”, ”SR” sub-columns give the abstraction computation, model checking, simulation and refinement time, respectively. ”Preds” gives the total number and the maximum number of predicates tracked at any program location. ”I” sub-column gives the number of abstraction refinement iterations.

5.1 Use of Octagonal Invariants during Predicate Abstraction and Refinement

Table 1 presents a comparison between three different implementations of the predicate abstraction and refinement loop. The ”Default” column uses the localization of predicates as described in [17]. This means that instead of maintaining a global set of predicates, localized predicates relevant to various basic blocks of the program are discovered by weakest pre-condition propagation along infeasible program traces.

The ”Strengthen” column uses the same framework as the ”Default” technique. However, it uses the strengthened transition relation for each basic block in the abstraction refinement loop. The strengthening is carried out using the octagonal invariants, which are pre-computed using the octagon abstract domain. We use control flow based packing for invariant generation and InvSelect heuristic for invariant selection (Section 4). Generation of octagonal invariants took five seconds for the `TCAS` benchmark. The ”BLAST” column presents the results of running the BLAST [1] software model checker with the Craig interpolation [15] options `craig2` and `predH7`.

The ”Time” sub-column presents the total time taken by the abstraction and refinement loop when checking a given property. For the ”Default” and ”Strengthen” techniques the breakup of total time (”Time”) is presented in the ”Abs”, ”MC”, and ”SR” sub-columns. The ”Abs” sub-column gives the total time spent in computing the predicate abstraction, the ”MC” sub-column is the total time spent in model checking the abstracted program, the ”SR” sub-column is the total time spent on the simulation of abstract counterexamples and refinement. The ”Preds” sub-column provides two numbers separated by a slash: 1) Total number of predicates present in the last iteration of abstraction refinement loop. 2) Maximum number of predicates tracked at a given program location. The ”Cex” sub-column provides the average length of infeasible traces that were given to the refinement procedure for generating new predicates. The ”I” sub-column gives the total number of abstraction refinement iterations.

Benchmark	Default						Strengthen					
	Time	Abs	MC	SR	Preds	I	Time	Abs	MC	SR	Preds	I
`SW-1`	29.1	8.3	2.3	18.5	53/17	14	**9.3**	2.9	0.5	5.9	16/4	6
`SW-2`	42.4	10.5	3.5	28.4	53/17	14	**9.1**	2.8	0.5	5.8	16/4	6
`SW-3`	**1.9**	0.8	0.3	0.8	16/14	5	3.0	0.8	0.3	1.9	16/14	5
`SW-4`	109.4	94	4.8	10.6	58/22	11	**6.3**	2.6	0.0	3.7	11/4	3

Table 2. Results on some industrial examples. Refer Table 1 for the meaning of various columns.

Reduction in the number of predicates: Observe that the strengthened transition relation ("Strengthen") allows checking the given properties with fewer predicates (first number in "Preds" column) on 9 out of 10 properties. Since all the three implementations use localization of predicates, the size of the abstract models produced can be exponential in the maximum number of predicates tracked at any program location. This is the second number in "Preds" column and it is smallest for the "Strengthen" column on 9 out of 10 properties as compared to both "Default" and "BLAST". As a result, the total time spent on model checking the abstractions ("MC") is smaller by 55% on average when using the strengthened transition relation as compared to the "Default" technique.
Reduction in the length of infeasible traces: The "Cex" column shows the average length of infeasible traces that were given to the refinement procedure. This number is consistently smaller when using the strengthened transition relation as compared to the "Default" technique. When refining an infeasible trace consisting of basic blocks $b_1, \ldots, b_k$, new predicates are discovered at each basic block b_i by the refinement procedure [15, 17]. Smaller infeasible traces were refined in the "Strengthen" case leading to fewer predicates as compared to the "Default" case.
Impact on running time: The significant reduction in the model checking time, enables "Strengthen" to outperform other techniques ("Default" and "BLAST") in terms of total time ("Time") on a majority of properties.
Results on `SW-*` *benchmarks:* We checked these benchmarks for standard property violations using "Default" and "Strengthen" techniques. Since the standard property checks are added automatically through control flow graph modification, a comparison with BLAST was not possible. The results on some `SW-*` benchmarks are summarized in Table 2. The meaning of the various columns in Table 2 is the same as in Table 1. We observed a reduction in the total number of abstraction refinement iterations, predicates needed, overall runtime as compared to "Default" on many `SW-*` benchmarks.

5.2 Generation of Invariants

We describe results for the two different octagon packing techniques discussed in Section 4.1. For both basic block based packing and control flow based packing we limit the size of each octagon pack to 10. That is no more variables are added to an octagon pack once its size exceeds 10. Table 3 presents the comparison between the basic block based packing and control flow based packing and their impact on the invariant generation. Only the results for some `SW-*` benchmarks are reported in this table.

The "BB" column gives the total number of basic blocks in the benchmark, the "Prop" column gives the total number of safety properties (reachability of labeled error statements, or automatically generated standard property monitors) in a benchmark. The "Block" column presents the results for the basic block based packing and the

Bench-mark	BB	Prop	Block				Control flow			
			Time	PackStats	Done	NumInv	Time	PackStats	Done	NumInv
tcas	224	10	18s	72/10/4.9	0	11196/5121	5s	49/5/2.7	0	3992/3456
SW-5	1587	295	190s	252/8/4.1	76	83478/38654	87s	180/6/1.5	90	35042/23431
SW-6	1986	592	264s	256/10/4.4	111	72972/50973	132s	203/6/1.5	131	58801/48612
SW-7	2440	542	576s	472/9/4.2	82	167738/87738	270s	310/9/1.5	82	105184/66130
SW-8	1472	402	237s	226/10/4.2	64	115254/90541	59s	132/8/2	64	98514/83096

Table 3. Comparison octagon packing techniques and their impact on invariant generation.

”Control flow” column presents results for the control flow based packing. The sub-column ”Time” gives the total time required to compute the invariants for the octagon packs generated using a given packing technique. The ”PackStats” column presents three numbers separated by a slash (/): total number of distinct octagon packs, maximum number of variables in an octagon pack, and average number of variables in an octagon pack. The ”Done” column shows the number of safety properties (”Prop” column) that can be proved by using the octagon invariants only. The ”NumInv” column presents two numbers separated by a slash (/): total number of invariants generated, and the total number of non-redundant invariants as computed by the octagon library [2].

Discussion of octagon packing results: The control flow based packing produces consistently less number of octagon packs as compared to the basic block based packing. This is expected as the number of octagon packs is proportional to the number of basic blocks in basic block based packing, and proportional to the number of conditions in the program in control flow based packing. The maximum and the average number of variables tracked in an octagon pack is smaller in the control flow based packing technique. Thus, the time taken to compute invariants using the control flow based packing is smaller (by 2.8$\times$ on average) as compared to the basic block based packing.

In order to compare the quality of invariants generated using the two packing techniques we did two experiments: First, we looked at the number of safety properties shown correct by the use of octagonal invariants themselves. This number is shown in the ”Done” column. We observed that the number of safety properties proved correct by basic block based packing was always a subset of or the same as those proved correct using control flow based packing.

Second, we used the generated invariants inside the predicate abstraction and refinement loop by transition relation strengthening. We found the addition of octagonal invariants generated (using either packing technique) to enable checking a given property with fewer predicates, as compared to not using the invariants. However, the addition of invariants generated using basic block based packing increased the predicate abstraction computation and simulation times significantly causing an overall increase in runtimes, as compared to not using invariants. For the TCAS benchmark after invariant generation and selection, an average of 8.6 invariants were added to each basic block when using the basic block based packing, as compared to an average of 1.9 invariants when using control flow based packing. As fewer invariants are added to each basic block with control flow based packing, the increase in abstraction computation and refinement times is much less as compared to using the basic block based packing. Overall, the addition of invariants generated using control flow based packing reduces the total runtime as compared to not using the invariants as discussed in Table 1, 2.

Bench	Default			InvSelect		
-mark	Tot	Max	Avg	Tot	Max	Avg
tcas	3456	24	15.4	441	12	1.9
SW-5	23431	43	18	2825	14	2.2
SW-6	48612	34	20.7	3307	8	1.4
SW-7	66130	58	23.4	5068	14	1.8
SW-8	83096	73	56.5	14844	31	10.1

Table 4. Application of InvSelect heuristic for selecting the invariants used for strengthening.

Why control flow based packing is useful: In many tools the generation of new predicates for abstraction refinement is done by computing the weakest pre-conditions of the conditions present in the control flow graph. Suppose the weakest pre-condition of a condition c for a certain number of steps results in predicates $p_1, \ldots, p_n$. Let $pvars$ denote the set of variables appearing in the predicates $p_1, \ldots, p_n$ and condition c. Let $vars(c)$ denote the octagon pack corresponding to condition c in the control flow based packing. If the size of $vars(c)$ is not restricted, then it is the case that $pvars \subseteq vars(c)$. Thus, the octagon packs computed using control flow based packing tend to cluster those variables for which relationships will be discovered later (through refinement) as new predicates and their weakest pre-conditions. Eagerly computing the relationships for such clusters and using them in the predicate abstraction and refinement loop, thus, attempts to get most benefit out of the efficiently computable invariants.

5.3 Invariant Selection for Strengthening

After invariant generation there can be many octagonal invariants that hold at each program location. As argued in Section 4.2, using all invariants that hold at program location l to strengthen the transition relation at l may not be beneficial. We apply a heuristic to filter out invariants that are not deemed important for checking a given property. The impact of the invariant selection heuristic InvSelect (Section 4.2) on the number of invariants that get selected for strengthening is summarized in Table 4. The ”Default” column shows the statistics before InvSelect selection heuristic is applied. The ”InvSelect” column gives the statistics after InvSelect selection heuristic is applied. The sub-column ”Tot” gives the total number of invariants that get selected, the ”Max” sub-column gives the maximum number of invariants selected at a basic block, and the ”Avg” sub-column gives the average number of invariants selected at a basic block.

The invariant selection heuristic InvSelect (Section 4.2) helps in reducing the number of invariants that get selected at each basic block for transition relation strengthening. For the TCAS benchmark, application of the InvSelect heuristic reduces the average number of invariants available for strengthening a given basic block from 15.4 to 1.9.

6 Conclusion

In this paper we presented how efficiently computable invariants can be used to improve the counterexample-guided abstraction refinement flow such as used in software verification tools using predicate abstraction. The invariants at program location l are *selectively* added to the concrete transition relation at l to obtain a *strengthened* transition relation at l. Using a strengthened transition relation in the predicate abstraction and refinement loop can lead to the creation of more precise abstract models leading to fewer and shorter infeasible traces. This can allow checking a given property with fewer predicates. More importantly, this technique can help in checking properties where using the standard predicate abstraction and refinement loop alone will take too long to converge (for example, properties depending on loop invariants). In our experiments we found a consistent reduction in the total number of predicates, maximum number of predicates tracked at a given program location, and the overall verification time.

References

1. BLAST tool, http://embedded.eecs.berkeley.edu/blast/.
2. Octagon abstract domain library, http://www.di.ens.fr/∼mine/oct/.
3. T. Ball, B. Cook, S.K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer-Aided Verification (CAV)*, pages 457–461, 2004.
4. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122, 2001.
5. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *CAV*, pages 319–331, 1998.
6. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and Veith H. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer-Verlag, 2000.
8. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, Sep–Nov 2004.
9. M. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV*, pages 293–304, 1998.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
11. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astreé analyzer. In *ESOP*, pages 21–30, 2005.
12. J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *FSE*, 2005.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
15. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
16. F. Ivančić, I. Shlyakhter, A. Gupta, Malay K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-SOFT. In *ICCD*. IEEE, 2005.
17. H. Jain, F. Ivančić, A. Gupta, and M.K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, pages 397–412, 2005.
18. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920, pages 459–473. Springer, 2006.
19. R.P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
20. S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt and F. Somenzi, editors, *CAV*, number 2725 in LNCS, pages 141–153, 2003.
21. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. http://www.di.ens.fr/∼mine/publi/article-mine-ast01.pdf.
22. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, December 2004.
23. H. Saïdi. Modular and incremental analysis of concurrent software systems. In *ASE*, 1999.
24. S. A. Seshia and R. E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *LICS*, pages 100–109, 2004.