Refining Approximations in Software Predicate Abstraction

Thomas Ball[1], Byron Cook[1], Satyaki Das[2], and Sriram K. Rajamani[1]

[1] Microsoft Corporation
[2] Stanford University

Abstract. Predicate abstraction is an automatic technique that can be used to find abstract models of large or infinite-state systems. In tools like SLAM, where predicate abstraction is applied to software model checking, a number of heuristic approximations must be used to improve the performance of computing an abstraction from a set of predicates. For this reason, SLAM can sometimes reach a state in which it is not able to further refine the abstraction.

In this paper we report on an application of Das & Dill's algorithm for predicate abstraction refinement. SLAM now uses this strategy lazily to recover precision in cases where the abstractions generated are too coarse. We describe how we have extended Das & Dill's original algorithm for use in software model checking. Our extension supports procedures, threads, and potential pointer aliasing. We also present results from experiments with SLAM on device driver sources from the Windows operating system.

1 Introduction

Automatic iterative abstraction refinement strategies allow us to apply model checking to large or infinite-state systems. When applying iterative abstraction refinement, we must strike a delicate balance between the accuracy of the abstraction that is produced and the speed at which it is generated. Therefore, in practice, a number of coarse approximations are typically employed. SLAM [1], for example, uses the *Cartesian approximation* [2] and the *maximum cube length approximation* [3]. In addition, SLAM's abstraction module uses a special purpose symbolic theorem prover which is similar to the work of Lahiri, Bryant & Cook [4] but optimized for speed and not precision. For this reason SLAM can run into the following situation:

- The abstract program has an error trace t.
- The trace t is found to be infeasible in the concrete program, and a set of predicates S are generated to rule out t in the abstraction.
- Even after abstracting the program with respect to S, since the abstraction is imprecise, t is still feasible in the abstraction.

As a result, iterative refinement is unable to make further progress. This situation, which we call an NDF-state (for "No Difference Found" in the error traces

K. Jensen and A. Podelski (Eds.): TACAS 2004, LNCS 2988, pp. 388–403, 2004.

seen in two successive iterations of iterative refinement), is not merely a theoretical concern. Users of predicate abstraction based tools do encounter it in practice.

```
1)          void main()
2)          {
3)              int x = *;
4)              int y = *;
5)              int z = *;
6)              if (x<y) {
7)                  if (y<z) {
8)                      if (!(x<z)) {
9)                          error();
10)                     }
11)                 }
12)             }
13)         }
```

Fig. 1. Example program

As an example, consider the small C program in Fig. 1, where `*` models nondeterministic value introduction. This program calls the procedure `error()` if `x` is not less than `z` after first checking that `x` is less than `y` and that `y` is less than `z`. By the transitivity of `<`, we know that `error()` will never be called. Without using the technique described in this paper, SLAM's standard predicate abstraction algorithm can not prove that the call to `error()` is unreachable. In this case, the incompleteness is due to SLAM's use of the Cartesian approximation, in which the update of each predicate is modeled independently.

To see why the property cannot be proved, consider the abstraction in Fig. 2. This is what SLAM would produce when given the following predicates: $\{\texttt{b1} \triangleq \texttt{x} < \texttt{y}, \texttt{b2} \triangleq \texttt{y} < \texttt{z}, \texttt{b3} \triangleq \texttt{x} < \texttt{z}\}$.

Notice how the abstraction models the update of the Boolean variables independently. For example, when modeling the effect of the instruction at line 5, SLAM determines that both `b2` and `b3` could be affected. It therefore assigns nondeterministically chosen values to the Boolean variables. In this example the approximation causes the abstraction to lose the transitivity correlation between `b1`, `b2` and `b3`. For this reason we cannot rule out the trace to the call to `error()` in the abstraction. A more precise modeling of the assignment would not lose this information. The more accurate translation might be as follows:

```
b2 = *;
if (b1 && b2) { b3 = 1; } else { b3 = *; }
```

Frequently, the style of approximation used in Fig. 2 is good enough to prove properties correct. In fact: it is often better — as argued by Ball, Podelski & Rajamani [2] — because the more precise abstraction comes with an unnecessary

```
 1)        void main()
 2)        {
  )            bool b1, b2, b3;
 3)            b1,b3 = *,*;
 4)            b1,b2 = *,*;
 5)            b2,b3 = *,*;
 6)            if (b1) {
 7)                if (b2) {
 8)                    if (!b3) {
 9)                        error();
10)                    }
11)                }
12)            }
13)        }
```

Fig. 2. Abstraction of the program in Fig. 1 with respect to the predicates {b1 $\triangleq$ x < y, b2 $\triangleq$ y < z, b3 $\triangleq$ x < z}. Note that b1,b2 = e,g; represents parallel assignment.

performance cost in cases where it is not needed. But, as we see in this case, the approximation can be too coarse at times.

The tension is that, if we model all possible correlations, the performance of computing the abstraction becomes intractable. However, for every correlation that we drop, there will inevitably be an example that requires it. What we need is a technique that refines these correlations only in the unusual cases in which they are important.

In this paper we describe such a technique. It is based on Das & Dill's algorithm for abstraction-refinement [5]. In the example described above, the technique would examine the trace to the call to `error()` and then, as a result of its analysis, refine the abstraction of the assignment to `z` as:

```
b2,b3 = *,* constrain (!(b1 && b2' && !b3'));
```

where the `constrain` keyword in the abstraction will cause SLAM's symbolic reachability module to ignore transitions that do not maintain the invariant. Notice that we're using `b3'` to represent the value of `b3` after the transition. With this refinement on the transition relation of the abstraction, a symbolic reachability engine can then be used to prove that the abstraction does not call `error()`.

This paper makes two novel contributions:

- We provide details on our extension to Das & Dill's original technique that allows for its application in software model checking. The extension that we describe supports procedures, threads and potential pointer aliasing.
- We demonstrate the technique's effectiveness on real-world industrial benchmarks. We report on the results of experiments run on the source code of a number of device drivers written for the Windows operating system. We demonstrate that the technique's impact on SLAM's accuracy is large, while its tax on performance is modest.

The paper is organized as follows: Section 2 describes SLAM's implementation of predicate abstraction with iterative refinement for software. In Section 3 we introduce CONSTRAIN, which is our extension to Das & Dill's algorithm. Section 4 describes the results of our experiments with CONSTRAIN and the checking of Windows operating system device drivers. We then relate our work to the literature in Section 5, discuss ideas for future work in Section 6, and make several concluding observations in Section 7.

2 Predicate Abstraction and Software Model Checking

Predicate abstraction is a technique proposed by Graf & Saïdi [6] and Colón & Uribe [7]. The idea is to find a set of predicates that must be maintained when proving a property of a transition system. The predicates are then used to compute an abstraction of the original transition system that is more amenable to algorithmic verification techniques such as model checking. Counter-example guided refinement is a technique that can be used to automate the search for these predicates.

SLAM is an implementation of counter-example driven refinement for the C language, using predicate abstraction. SLAM combines three modules to automatically compute an abstraction of a C program based on counter-examples encountered during the abstraction process. The modules are called C2BP [2,3], BEBOP [8,9] and NEWTON [10]

C2BP is used within SLAM to abstract C programs. C2BP takes as input a program P and a set of predicates $\{e_1, \cdots, e_n\}$, which are expressions with Boolean type that may contain relational and arithmetical sub-expressions, and produces an abstraction. The abstraction is represented as a *Boolean program*—a form much like a C program in which all variables have type `bool`. C2BP operates by abstracting each statement s in P into a corresponding statement s_b in the abstraction. The statement s_b conservatively maintains the value of each predicate e_i using a representative Boolean variable.

SLAM performs symbolic model checking on the Boolean program produced by C2BP with BEBOP. If BEBOP can prove that the Boolean program does not call `error()` then the C program also can not call `error()`. We know this is true because the Boolean program is an abstraction of the C program. If BEBOP finds a failing trace in the abstraction that corresponds to a real error in the original C program then SLAM terminates after providing information to the user about the trace. Otherwise, the spurious counter-example found by BEBOP is used to compute additional predicates. The tasks of determining if found traces are real and the generation of new predicates is performed by the module NEWTON.

In the case that the counter-example is spurious and additional predicates are found, then C2BP is called again. This loop is repeated until the program is proved correct, a real trace to `error()` is found, or a state is reached in which a spurious trace cannot be ruled out via additional predicates. As mentioned in Section 1, this final case is an NDF-state.

3 Extending Das and Dill's Abstraction Refinement for Software Model Checking

Das & Dill's algorithm [5] refines an abstract transition relation by looking for spurious transitions in traces and ruling them out in the abstraction using only the predicates already available. Our adaptation of Das & Dill's algorithm to software model checking is called CONSTRAIN. CONSTRAIN takes, as arguments, the original C program and a trace through the abstraction that includes the valuation of the predicates at each step. CONSTRAIN then returns a set of constraints to be added to the abstraction. In pseudo ML-syntax, CONSTRAIN has the following type:

$$\text{CONSTRAIN} : (\texttt{pgm} \times \texttt{abstr} \times \texttt{trace}) \rightarrow \texttt{constraint list}$$

CONSTRAIN analyzes each abstract transition in the trace using a function called SPURIOUS, which has the following ML-like type:

$$\text{SPURIOUS} : (\texttt{inst} \times \texttt{absState} \times \texttt{absState} \times (\texttt{absState option})) \rightarrow \texttt{query}$$

In the invocation $\text{SPURIOUS}(i, a_0, a_1, \texttt{Some}(a_c))$, the first parameter ($i$) is the transition's instruction in the C program. The second and third parameters (a_0 and a_1) are the transition's pre-state and post-state. The fourth parameter a_c is optional, and is used only for processing procedure return instructions. For a return instruction from procedure R, a_c is the state that the abstraction was in just before calling the procedure R. For other instructions, the fourth parameter is passed the value `None`. CONSTRAIN maintains a stack on which the abstract state is pushed when a trace crosses into a procedure call, and is popped when a trace crosses a procedure return transition, so as to supply the fourth parameter for SPURIOUS for processing return instructions.

The type `query` is a logic expression that CONSTRAIN can then pass to an automatic theorem prover such as Simplify [11], Verifun [12] or CVC [13]. If the query is provably valid then the abstract transition at instruction i is spurious and can be constrained.

3.1 Discovering Spurious Transitions

SLAM represents C programs in an intermediate nondeterministic control-flow-graph representation where all instructions are either assignment statements, assume statements, procedure calls or procedure returns. We define SPURIOUS for each of these instruction types. Let $\mathcal{E}$ be the concretization function from abstract states to sets of concrete states, and let $\texttt{Pre}(i, \Theta)$ be the pre-image of states Θ with respect to instruction i. More precisely,

$$\texttt{Pre}(i, \Theta) \triangleq \{c \mid \exists c' \in \Theta \text{ such that } c \text{ can transition to } c' \text{ via instruction } i\}$$

For deterministic instructions, `Pre` can be computed using `WP`, the weakest precondition [14]. Within `WP` we use Morris's general axiom of assignment [15] to conservatively model possible pointer aliasing relationships.

Assignment and Assume statements. If i is an assignment statement or an assume statement, then SPURIOUS is defined as:

$$\text{SPURIOUS}(i, a_0, a_1, \texttt{None}) \triangleq (\mathcal{E}(a_0) \implies \neg \texttt{Pre}(i, \mathcal{E}(a_1)))$$

If $\text{SPURIOUS}(i, a_0, a_1, \texttt{None})$ is a valid query then a constraint $\neg(a_0 \wedge a_1')$ can be added to the abstraction to rule out the spurious abstract transition. Here a_1' denotes the state a_1 expressed in terms of primed or next-state variables.

Consider the example from Figs. 1 and 2 from Section 1. Suppose that we call SPURIOUS with the abstract transition

$$\texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3} \quad \overset{5}{\dashrightarrow} \quad \texttt{b1} \wedge \texttt{b2} \wedge \neg\texttt{b3}$$

That is, we are asking SPURIOUS the question: "Should the abstraction's state be allowed to change from $(\texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3})$ to $(\texttt{b1} \wedge \texttt{b2} \wedge \neg\texttt{b3})$ by executing the statement at line 5?"

More precisely, we are interested in the value of:

$$\text{SPURIOUS}(5,\ \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3},\ \texttt{b1} \wedge \texttt{b2} \wedge \neg\texttt{b3},\ \texttt{None})$$

Note that $\mathcal{E}(\texttt{b1}) \triangleq \texttt{x} < \texttt{y}$, $\mathcal{E}(\texttt{b2}) \triangleq \texttt{y} < \texttt{z}$, and $\mathcal{E}(\texttt{b3}) \triangleq \texttt{x} < \texttt{z}$. Let us use α to denote the nondeterministically chosen value that was introduced by $*$. After unfolding Pre and $\mathcal{E}$, $\text{SPURIOUS}(5,\ \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3},\ \texttt{b1} \wedge \texttt{b2} \wedge \neg\texttt{b3},\ \texttt{None})$ equals

$$((\texttt{x} < \texttt{y}) \wedge (\texttt{y} < \texttt{z}) \wedge (\texttt{x} < \texttt{z})) \Rightarrow (\neg((\texttt{x} < \texttt{y}) \wedge (\texttt{y} < \alpha) \wedge \neg(\texttt{x} < \alpha)))$$

Since this query can be proved valid by a theorem prover, CONSTRAIN then produces the following constraint which rules out this transition:

```
constrain (!(b1 && b2 && b3 && b1' && b2' & !b3'));
```

Notice that this constraint is not quite the same as the constraint from Section 1. Later, we will describe an optimization that produces that constraint.

Procedure calls and returns. Let instruction i be a call to procedure R from some procedure Q of the C program:

$$x = R(e_1, e_2, \ldots, e_N)$$

For convenience, we will make the procedure call and the assignment of the return value explicit in the intermediate representation of the C program with the introduction of a fresh variable x_{ret}:

$$x_{ret} = R(e_1, e_2, \ldots, e_N);$$
$$x = x_{ret};$$

The abstraction of this call is divided into three parts: (1) the computation of actual parameters, (2) the corresponding call in the Boolean program, and (3) computation of the side effects of the call. The computation of the actual

parameters in the Boolean program takes the form: $p_1, p_2, \ldots, p_j = c_1, c_2, \ldots, c_j$, as defined by Ball *et al.* [3], where j is the number of formal parameters for function R in the Boolean program (note that j could be different from N, the number of formal parameters for R in the C program), and $\{p_1, p_2, \ldots p_j\}$ are temporary variables introduced in the Boolean program. The actual call to R in the Boolean program takes the form:

$$ret_1, ret_2, \ldots, ret_k = R(p_1, p_2, \ldots, p_j)$$

where k is the number of predicates in R that are return predicates, and the variables $\{ret_1, ret_2, \ldots, ret_k\}$ are temporaries used to hold values of the return predicates in Q. Immediately following the call, all local predicates that could be modified by the call are updated.

Let a_0 be the abstract state immediately before execution of i, and let a_1 be the abstract state just after the call is made. Let the formal parameters of f be $f_1, f_2, \ldots f_n$. Let i^p denote the special parallel assignment instruction $f_1, f_2, \ldots f_n = e_1, e_2, \ldots e_n$. SPURIOUS is defined as:

$$\text{SPURIOUS}(i, a_0, a_1, \texttt{None}) \triangleq \mathcal{E}(a_0) \Longrightarrow \neg \texttt{Pre}(i^p, \mathcal{E}(a_1))$$

Assume that CONSTRAIN uses a function δ to map formal parameter predicates of R to the corresponding parameter temporaries in $\{p_1, p_2, \ldots p_j\}$, If SPURIOUS$(i, a_0, a_1, \texttt{None})$ is valid, then CONSTRAIN will add $\neg(a_0 \wedge \delta(a_1'))$, to the computation of actual parameters for the call in the abstraction.

We will now define SPURIOUS for procedure return instructions. Let i be the statement `return r` in R. Recall that the call to R in Q assigns the return value to variable x. We first define two auxiliary functions: ρ over the variables in scope at Q, and γ over variables in scope at R. The function ρ is defined as follows:

$$\rho(v) = \begin{cases} v & \text{if } v \text{ is a local variable or formal parameter of } Q \\ v_o & \text{if } v \text{ is a global variable, where } v_o \text{ is a fresh variable} \end{cases}$$

Conceptually, we can think of v_o as caching the value of global v in Q just before the call to R. The function γ is defined as follows:

$$\gamma(v) = \begin{cases} v & \text{if } v \text{ is global} \\ x_{ret} & \text{if } v \text{ is } r \\ \rho(e_i) & \text{if } v \text{ is a symbolic value of the parameter } f_i \text{ to } R \end{cases}$$

Let i^r denote the assignment statement in the intermediate form $x = x_{ret}$. Let a_0 be an abstract state just before execution of the return statement, and let a_1 be an abstract state just after execution of the return statement, and let a_c be the state just before execution of the call to R. In this case we define SPURIOUS as:

$$\text{SPURIOUS}(i, a_0, a_1, \texttt{Some}(a_c)) \triangleq (\gamma(\mathcal{E}(a_0)) \wedge \rho(\mathcal{E}(a_c))) \Longrightarrow \neg \texttt{Pre}(i^r, \mathcal{E}(a_1))$$

If SPURIOUS$(i, a_0, a_1, \texttt{Some}(a_c))$ is valid, then CONSTRAIN will add $\neg(a_0 \wedge \omega(a_c) \wedge a_1')$ to the return transition in the abstraction, where ω maps each predicate in R to the corresponding return temporary from $\{ret_1, ret_2, \ldots, ret_k\}$, is added for the instruction that computes the side-effects of the call.

```
1)    int foo(int a)                    11)    void main()
2)    {                                 12)    {
3)        int b = *;                    13)        int x = *;
4)        if (a<b) {                    14)        int y = *;
5)            return b;                 15)        int z = *;
6)        } else {                      16)        if (x<y) {
7)            exit();                   17)            z = foo(y);
8)        }                             18)            if (!(x<z)) {
9)    }                                 19)                error();
10)                                     20)            }
                                        21)    }
```

Fig. 3. Example program with procedure calls and returns

3.2 Example of Return Processing

As an example of how CONSTRAIN and SPURIOUS handle procedure returns, consider the program in Fig. 3. When modeling the return from `foo` we must consider three states: (1) The state of the program just before the call to `foo`, (2) the state of the program just before the return from `foo`, and (3) the state of the program just after the return from `foo`.

Imagine that SLAM is tracking the following predicates (where α is used to represent the symbolic initial value of the argument to `foo`):

$$\begin{aligned} \texttt{main} &: \{\texttt{b1} \triangleq \texttt{x} < \texttt{y}, \texttt{b2} \triangleq \texttt{y} < \texttt{z}, \texttt{b3} \triangleq \texttt{x} < \texttt{z}\} \\ \texttt{foo} &: \{\texttt{a1} \triangleq \alpha < \texttt{b}\} \end{aligned}$$

The following is part of a spurious trace that BEBOP might find while model checking the abstraction:

$$\texttt{b1} \wedge \neg\texttt{b2} \wedge \neg\texttt{b3} \xrightarrow{17} \cdots \longrightarrow \texttt{a1} \xrightarrow{5} \texttt{b1} \wedge \neg\texttt{b2} \wedge \neg\texttt{b3}$$

At the procedure call transition at line 17, CONSTRAIN will push the state $\texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}$ onto the stack. Then, when processing the return instruction at line 5, CONSTRAIN will pop the stack and call SPURIOUS with this popped state as the optional parameter:

$$\textsc{Spurious}(17,\ \texttt{a1},\ \texttt{b1} \wedge \neg\texttt{b2} \wedge \neg\texttt{b3},\ \texttt{Some}(\texttt{b1} \wedge \neg\texttt{b2} \wedge \neg\texttt{b3}))$$

which we can evaluate by unfolding the definitions of $\mathcal{E}$, γ, ρ and SPURIOUS:

$$(\gamma(\mathcal{E}(\texttt{a1})) \wedge \rho(\mathcal{E}(\texttt{b1} \wedge \neg\texttt{b2} \wedge \neg\texttt{b3}))) \Longrightarrow \neg\texttt{Pre}(17^r, \mathcal{E}(\texttt{b1} \wedge \neg\texttt{b2} \wedge \neg\texttt{b3}))$$

$$\Leftrightarrow_{\mathcal{E}} \quad \gamma((\alpha < \texttt{b})) \wedge \rho((\texttt{x} < \texttt{y}) \wedge \neg(\texttt{y} < \texttt{z}) \wedge \neg(\texttt{x} < \texttt{z})) \Longrightarrow \neg\texttt{Pre}(17^r, (\texttt{x} < \texttt{y}) \wedge \neg(\texttt{y} < \texttt{z}) \wedge \neg(\texttt{x} < \texttt{z}))$$

$$\Leftrightarrow_{\rho} \quad \gamma((\alpha < \texttt{b})) \wedge (\texttt{x} < \texttt{y}) \wedge \neg(\texttt{y} < \texttt{z}) \wedge \neg(\texttt{x} < \texttt{z}) \Longrightarrow \neg\texttt{Pre}(17^r, (\texttt{x} < \texttt{y}) \wedge \neg(\texttt{y} < \texttt{z}) \wedge \neg(\texttt{x} < \texttt{z}))$$

$$\Leftrightarrow_{\texttt{Pre}\&\gamma} \quad (\texttt{y} < \texttt{z}_{\texttt{ret}}) \wedge (\texttt{x} < \texttt{y}) \wedge \neg(\texttt{y} < \texttt{z}) \wedge \neg(\texttt{x} < \texttt{z}) \Longrightarrow \neg((\texttt{x} < \texttt{y}) \wedge \neg(\texttt{y} < \texttt{z}_{\texttt{ret}}) \wedge \neg(\texttt{x} < \texttt{z}_{\texttt{ret}}))$$

Since the last expression is provably valid[1] CONSTRAIN will add the constraint

```
constrain ( !(b1 && !b2 && b3 && ret1 && b1' && !b2' & !b3'));
```

(where `ret1` is the return temporary corresponding to `a1`) to the instruction that computes the side effects of the call.

3.3 Optimizations and Additional Extensions

In the example from Section 3.1 we were not able to compute the constraint promised in Section 1. However, by iteratively dropping predicates from the abstract states and re-evaluating SPURIOUS we can find this strengthened constraint. After searching for the necessary predicates we will find that

$$\text{SPURIOUS}(5,\ \texttt{b1},\ \texttt{b2} \wedge \neg \texttt{b3},\ \texttt{None})$$

is valid, which gives us the following stronger constraint `!(b1 && b2' && !b3')`. This is an important optimization because it allows CONSTRAIN to eliminate more spurious transitions than the weaker constraint does. However, the approach that we currently use to compute this strengthening is not optimal. We will discuss this further in Section 6.

As another optimization consider the fact that, although it is not always true, the constraint that we found in Section 1 can be encoded as something that holds for all time. That is, in this case, transitivity is true regardless of the transition at line 5. For this reason, the constraint could be expressed in such a way that the reachability engine ignores all transitions that violate the invariant whenever the invariant is in scope. As an optimization, BEBOP provides exactly this construct through the keyword `enforce`. When performing reachability analysis on this abstraction, BEBOP will then prune states from consideration in which the `enforce` constraint is not true.

CONSTRAIN can potentially find and rule out spurious transitions at multiple points along a given trace. Through experimentation we have found that, when CONSTRAIN is used lazily, the overall performance of SLAM is at its best when CONSTRAIN is allowed to return only up to the first five constraints found during a single pass. We have also found that, when an `enforce` constraint is found, it is best to return immediately.

CONSTRAIN can also find multiple constraints if it is executed multiple times using the same set of predicates, but different traces. That is, if CONSTRAIN and BEBOP are alternated without the addition of new predicates via NEWTON. We have found that, when used lazily, it is most efficient to allow up to four alternations of BEBOP and CONSTRAIN before running NEWTON again in order to find additional predicates. Beyond these four calls, few of the constraints that are found and instrumented into the abstraction seem to contribute to SLAM's overall accuracy or performance.

[1] By the transitivity of $<$ over `x`, `y` and $\texttt{z}_{\texttt{ret}}$

Constrain can easily be extended to support thread-switches in the traces that are passed to it. For this purpose Constrain maintains a set of call-stacks—one for each thread. The traces contain information about when threads are switched. Constrain then switches between the stacks during the analysis.

4 Experimental Results

In Section 1 we conjectured that Das & Dill's algorithm [5] could be used to improve Slam's precision while not significantly degrading performance. In this section we attempt to measure this with experimental findings. We also demonstrate how several of C2bp's approximations effect Slam's performance and accuracy.

4.1 Improving Accuracy with Das and Dill's Algorithm

To determine the effectiveness of Constrain, we have executed Slam on software model checking benchmarks using the following three configurations:

No Constrain: Slam without support from Constrain.

Lazy mode: Slam using Constrain, in which Constrain is used simply as an NDF-state recovery method. Each call to Constrain is allowed to generate up to five constraints, and Constrain is called up to four times before returning to the standard Slam loop.

Eager mode: Slam with Constrain in which C2bp is configured to be maximally imprecise, leaving Constrain as the only working abstraction mechanism. In this configuration, because Constrain is doing all of the reasoning during the abstraction computation, each call to Constrain is allowed to add up to 200 new constraints, and Constrain can be executed up to 200 times before new predicates are sought out.

For each of these three configurations, Slam was then used to check 35 safety properties of 26 Windows device drivers (910 checks in total, 644 of which require Slam's analysis). The timeout threshold was set to 1200 seconds. The memory threshold was set to 500 megabytes. The sizes of the device drivers used in these experiments ranged from 10,000 to 40,000 lines of C code. The results are in Fig. 4.

The most important aspect of this figure is the first row, which represents the number of cases in which Slam terminated in an NDF-state. In the **No Constrain** column we can see that in 170 out of the 910 checks (or nearly one in five checks if we ignore the 266 cases where a property did not apply to a driver) Slam became stuck in an NDF-state. This was the result that Slam's users experienced before we implemented Constrain: 170 cases where Slam was not able to produce a useful result due to incompleteness.

The **Lazy mode** column represents the result of running Slam with Constrain as an NDF-state recovery method. In 167 out of 170 NDF-state cases, Constrain provided accuracy enough that Slam was able to make further

Result	No Constrain	Lazy mode	Eager mode
Termination in NDF-state	170	3	3
Property passes found	470	554	544
Time/memory threshold exceeded	59	107	122
Property violations found	19	50	45
Property not applicable to driver	266	266	266

Fig. 4. Results from experiments with CONSTRAIN on 26 device drivers and 35 safety properties. Of the 910 checks, only 644 require SLAM's analysis. The time threshold was set to 1200 seconds. The memory threshold was set to 500 megabytes.

progress. The three cases in which SLAM still reached an NDF-state are due to some subtle problems that occur when refining abstractions in the presence of some special heap-based data structures.

The **Eager** column displays the result from running SLAM using a disabled C2BP module: The functional result is largely the same, but the performance is less impressive than the **Lazy mode**. In this configuration, 15 additional checks exceed the 20 minute timeout.

4.2 Impact on Performance

In order to measure CONSTRAIN's effect on SLAM's performance we first recorded the total CONSTRAIN runtime required to compute the results of the **Lazy mode** column of Fig. 4 and compared it the total runtime of SLAM. We found that ten percent of SLAM's overall runtime was spent in the CONSTRAIN module.

Then, in order to determine the impact on SLAM's performance in just the cases where CONSTRAIN is actually used, we collected a number of averages while running SLAM in the default (lazy) configuration on 126 benchmarks that required at least one call to CONSTRAIN in order to recover from an NDF-state. The statistical averages are displayed in Fig. 5.

Description	Average
SLAM runtime	354.31s
Amount of runtime spent in CONSTRAIN	81.24s
Calls to CONSTRAIN	4.40
CONSTRAIN refinement iterations	1.74
Constraints generated	6.17

Fig. 5. Averages collected when applying SLAM to 126 device-driver based benchmarks in which CONSTRAIN is required to recover from an NDF-state.

We can see that, even when we limit the averages to those cases in which CONSTRAIN is called, its impact on SLAM's overall runtime is not overwhelming.

In these cases, 23 percent of SLAM's overall runtime was spent computing constraints. On average, CONSTRAIN was called about five times per model-checking run. And, on average, the refinement loop using CONSTRAIN and BEBOP was called in less than two iterations of SLAM's main loop. Finally, employing CONSTRAIN resulted in, on average, less than seven constraints to the final Boolean abstraction found.

4.3 Tuning the Approximations

In order to understand how the calls to CONSTRAIN are affected by C2BP's approximations we ran SLAM in various configurations on four drivers with six properties. In one case SLAM's analysis is not required.

Unfortunately, the Cartesian abstraction cannot be turned off in C2BP. However, we can adjust the use of the maximum cube length approximation and our fast but imprecise symbolic theorem prover.

Description	**S/3**	**S/5**	**S/∞**	**C/3**	**C/5**	**C/∞**
Total calls to CONSTRAIN	99	112	408	10	7	5
Total constraints generated	145	160	705	7	5	3
Termination in NDF-state	0	0	0	0	0	0
Property passes found	15	14	13	6	5	4
Time/memory threshold exceeded	7	8	9	17	18	19
Property violations found	1	1	1	0	0	0
Property not applicable to driver	1	1	1	1	1	1

Fig. 6. Totals collected while checking six properties on four drivers with SLAM using 6 different configurations. **S** = use of our symbolic theorem prover; **C** = use of a standard concrete theorem prover; 3 = maximum cube length set to three; 5 = maximum cube length set to five; ∞ = maximum cube length not set. The timeout threshold was set to 1200 seconds. The space threshold was set to 500 megabytes.

Fig. 6 displays the results from running SLAM in the following six configurations: **S/3** is SLAM's default configuration, in which the symbolic theorem prover is used and the cubes are limited to a maximum of three. **S/5** uses the symbolic theorem prover with cubes limited to five. **S/∞** uses the symbolic theorem prover and does not limit the cubes. **C/3** uses a more accurate concrete theorem prover and limits the cubes to three. **C/5** uses the concrete theorem prover and limits the cubes to five. **C/∞** uses the concrete theorem prover and does not limit the cubes.

In Fig. 6 **S/3** provides the best performance and accuracy because it maximizes the property violations and passes found, and minimizes the cases in which SLAM's execution exceeded the time or memory threshold. An interesting side-effect of enlarging the cube sizes when the symbolic theorem prover is used is

that the precision actually goes down. The reason is that this prover only implements a subset of first-order logic and larger cube sizes increase the chances of unsupported symbols.

Fig. 6 demonstrates that, rather than shutting off C2BP's approximations, it is best to leave them on for the common cases. In the uncommon cases that require more precision, CONSTRAIN can patch up the result.

5 Related Work

The use of counter-examples to automatically guide refinement in model checking has been described in many papers, including those by Alur *et al.* [16], Balarin & Sangiovanni-Vincentelli [17] and Clarke *et al.* [18]. Predicate abstraction combined with counter-example driven refinement has been since widely adopted for the verification of software-like systems. Examples of tools that use this strategy include InVeSt [19], SLAM [10], BOOP [20], CALVIN [21] Murϕ– [22], BLAST [21] and MAGIC [23]. However, in all of these tools, refinement has traditionally consisted of adding additional predicates.

Rather than simply adding more predicates, more recent research work has focused on improving the abstractions produced in predicate abstraction. The work in this area can be categorized as follows:

Removing corner-case problems: As an example consider the Cartesian abstraction. This approximation essentially causes invariants involving disjunctions of predicates to be lost. CONSTRAIN deals with this as it does with all other approximations: it explicitly restricts transitions one-by-one that violate the disjunctions. Another way of avoiding this is to add predicates which explicitly contain these disjunctions. In the context of BLAST, recent work by Henzinger, Jhala & McMillan [24] does precisely this.

Better abstraction algorithms: For example, Lahiri, Bryant & Cook [4] describe a technique that uses a symbolic theorem prover to avoid the necessity of the *maximum cube length approximation*. In principle, by using this theorem prover, C2BP could improve the quality of the abstractions generated. Because our symbolic prover supports such a limited subset of logic we have found (in Fig. 6) that the maximum cube length approximation is *still* a good thing to do.

Making better use of the predicates: Our work falls into this category. Another strategy in this category is the work on removing redundant predicates [25,23]. For example, during the predicate discovery phase, Chaki *et al.* [23] analyze many counter-example traces instead of one. They find predicates that can eliminate each of the spurious traces. Then they find a minimal set of these predicates so that each of the spurious traces can be removed and add those predicates.

6 Future Work

In Section 3.3 we alluded to the fact that Constrain attempts to strengthen the constraints that it finds. To compute this strengthening Constrain speculatively throws out predicates one-by-one from the calls to Spurious until the minimal set that is required is found. This results in an algorithm that makes a linear number of calls to the theorem prover for each constraint generated. However, many automatic theorem provers compute something that is close to the strengthening that Constrain is searching for. Examples include CVC [13] and Verifun [12]. In the future we should investigate using this information provided by the theorem prover to strengthen constraints.

NDF-states are currently the only trigger that cause Slam to invoke Constrain. It is possible that there are additional good triggers. For example: if a large majority of predicates are being added to rule out transitions through a small segment of code, Constrain might be better able to refine the abstraction than Newton and C2bp. Adding additional triggers is something that we hope to investigate further.

The `enforce` constraints mentioned in Section 3.3 are extremely strong and can, in theory, require the reachability engine to perform lots of unnecessary work. In the future we should explore other options. For example, we could simply add a standard constraint at the beginning of the function body where today we place an `enforce` constraint. This would mean that the constraint would not be maintained as an invariant by the reachability module, but would in many cases be maintained by the abstraction instead. Another alternative would be to add the time-invariant as a constraint at each transition along the trace passed to Constrain.

Constrain currently only supports forward passes through traces. However, in principle, Constrain could be run backwards along a trace. It could then quickly return only a small subset of constraints generated from the last spurious transitions. We have not yet investigated this approach. One complication to this method is that a backwards analysis of a trace by Constrain will have to handle procedure calls and returns in a special way. The problem is that, at the point where the return is being analyzed, we must know the state of the abstraction at which the procedure was called. In the forward-based analysis this is natural to track. But, in order to support a backwards-based analysis, we would need to add a forward pre-pass of the trace in which the states of the program just before procedure calls are recorded.

7 Conclusion

Predicate-abstraction tools do not scale without the use of some approximations. However, with increased imprecision, predicate-abstraction based refinement loops are not always able to make progress. In this paper we have described an application of and extension to Das & Dill's method, called Constrain, which selectively recovers precision in cases where it is needed. Constrain handles the characteristics of software, including procedures, threads, and pointer

aliasing. Our experimental results demonstrate that with CONSTRAIN, SLAM benefits from C2BP's fast approximations in the cases where they work, and recovers precision in the cases where the precision is needed to make progress. In practice, this strategy has tremendously improved the usability of SLAM, making it a predictable and usable tool for the validation of device driver correctness.

Acknowledgments. The authors would like to thank David Dill, Ranjit Jhala, Shuvendu Lahiri, Vladimir Levin, Ken McMillan, and Shaz Qadeer for their ideas and comments related to this work

References

1. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstractions for model checking C programs. In: TACAS 01: Tools and Algorithms for Construction and Analysis of Systems. LNCS 2031, Springer-Verlag (2001) 268–283
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI 01: Programming Language Design and Implementation, ACM (2001) 203–213
4. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 141–153
5. Das, S., Dill, D.L.: Successive approximation of abstract transition relations. In: Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science. (2001) June 2001, Boston, USA.
6. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: CAV 97: Conference on Computer Aided Verification. Volume 1254 of Lecture notes in Computer Science., Springer-Verlag (1997) 72–83 June 1997, Haifa, Israel.
7. Colón, M.A., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV 98: Conference on Computer-Aided Verification. Volume 1427 of Lecture Notes in Computer Science., Springer-Verlag (1998) 293–304
8. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130
9. Ball, T., Rajamani, S.K.: Bebop: A path-sensitive interprocedural dataflow engine. In: PASTE 01: Workshop on Program Analysis for Software Tools and Engineering, ACM (2001) 97–103
10. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research (2002)
11. Detlefs, D., Nelson, G., , Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
12. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 355–367

13. Stump, A., Barrett, C., Dill, D.: CVC: a cooperating validity checker. In: CAV 02: International Conference on Computer-Aided Verification. (2002) 87–105
14. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
15. Morris, J.M.: A general axiom of assignment. In: Theoretical Foundations of Programming Methodology. Lecture Notes of an International Summer School. D. Reidel Publishing Company (1982) 25–34
16. Alur, R., Itai, A., Kurshan, R., Yannakakis, M.: Timing verification by successive approximation. Information and Computation 118(1) (1995) 142–157
17. Balarin, F., Sangiovanni-Vincentelli, A.L.: An iterative approach to language containment. In: CAV 93: International Conference on Computer-Aided Verification. (1993) 29–40
18. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV 00: International Conference on Computer-Aided Verification. (2000) 154–169
19. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: TACAS 01: Tools and Algorithms for the Construction and Analysis of Systems. (2001)
20. Weissenbacher, G.: An abstraction/refinement scheme for model checking C programs. Master's thesis, Graz University of Technology, Graz, Austria (2003)
21. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: CAV 03: International Conference on Computer-Aided Verification, Springer Verlag (2003) 262–274
22. Das, S., Dill, D.L.: Counter-example based predicate discovery in predicate abstraction. In: FMCAD 02: Formal Methods in Computer-Aided Design, Springer-Verlag (2002)
23. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: CHARME 03: Advanced Research Working Conference on Correct Hardware Design and Verification Methods. (2003)
24. Thomas A. Henzinger, Ranjit Jhala, R.M., McMillan, K.L.: Abstractions from proofs. In: POPL 04: Symposium on Principles of Programming Languages, ACM Press (2004)
25. Clarke, E., Grumberg, O., Talupur, M., Wang, D.: Making predicate abstraction efficient: How to eliminate redundant predicates. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 355–367