Using Counterexample Analysis to Minimize the Number of Predicates for Predicate Abstraction

Thanyapat Sakunkonchak, Satoshi Komatsu, and Masahiro Fujita

VLSI Design and Education Center, The University of Tokyo
2–11–16 Yayoi, Bunkyo-ku, Tokyo, 113–0032, Japan
{thong,komatsu}@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract. Due to the success of the model checking technique in the hardware domain, over the last few years, model checking methods have been applied to the software domain which poses its own challenges, as software tends to be less structured than hardware. Predicate abstraction is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type. The cost for computation of abstraction and model checking depends on the number of state variables in the abstract model. In this paper, we propose a simple, yet efficient method to minimize the number of predicates for predicate abstraction. Given a spurious counterexample, at least one predicate is assigned at each program location during the refinement process. The computational cost depends proportionally to the number of assigned predicates. In this paper, instead, we search the counterexample to find the conflict predicates that caused this counterexample to be spurious. Then, we assign the necessary predicates to the abstract model. We compare the performance of our technique with the interpolation-based predicate abstraction tool like BLAST. The proposed method presents significantly better experimental results on some examples with large set of predicates.

1 Introduction

Model checking is the formal verification technique most-commonly used in the verification of RTL or gate-level hardware designs. Due to the success of the model checking technique in the hardware domain [1,2], over the last few years, model checking methods have been applied to the software domain, and we have seen the birth of software model checkers for programming languages such as C/C++ and Java.

There are two major approaches to software model checking. The first approach emphasizes *state space exploration*, where the state space of a system model is defined as the product of the state spaces of its concurrent finite-state components. The state space of a software application can be systematically explored by driving the "product" of its concurrent processes via a run-time scheduler through all states and transitions in its state space. This approach is developed in the tool Verisoft [3]. The second approach is based on *static analysis and abstraction* of software. It consists of automatically extracting a model

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 553–563, 2007.
© Springer-Verlag Berlin Heidelberg 2007

out of a software application by statically analyzing its code and abstracting away details, and then applying symbolic model checking to verify this abstract model.

In the context of the second approach, *predicate abstraction* [4,5,6] is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type while preserving the behaviors and control constructs of the original. The generated abstract model is conservative in the sense that for every execution in the original model there is a corresponding execution in the abstract model. Each predicate in the original model is corresponding to a Boolean variable in the abstract model. Abstraction is coarse in the beginning. The spurious counterexample from model checker gives an information to refine the abstract model to be more precise.

We address three problems on refinement of predicate abstraction.

- **Select new predicates:** The choice of selection of new predicates is important. At every refinement loop, spurious behaviors can be removed by adding new predicates to make the relationships between existing predicates more precise. New predicates from the wrong selection can cause the abstract model to be intractable during the refinement.
- **Scope of predicates:** When predicates are used globally, coping with relationship of all predicates through the entire model is impossible when the size of the abstract model is large. Hence, the idea of localizing predicates should be considered.
- **Number of predicates:** Although localization of predicates is utilized, the computation of predicate abstraction and model checking are exponential with the number of predicates. The cost of computation in abstraction process can be reduced drastically when the number of predicates is smaller.

There are many software model checking tools that apply predicate abstraction, e.g. SLAM [5], BLAST [6] or F-Soft [7]. The problems addressed above can be handled by these tools.

- SLAM uses the techniques called *Cartesian approximation* and *maximum cube length approximation* to reduce the number of predicates in each theorem prover query. The selection of new predicates is conducted by a separate refinement algorithm.
- In BLAST toolkit, besides *lazy abstraction* technique which does *on-the-fly* abstraction and refinement of abstract model, the new refinement scheme based on *Craig interpolation* [8] was proposed. With this method, localization for predicates and predicate selection problems can be handled.
- Recent work [7] describes localization and register sharing to reduce the number of predicates at each local part of the abstract model. *Weakest preconditions* are used to find new predicates. This approach has been implemented to the F-Soft toolkit.

This work is inspired by a localization and register sharing [7] which applied to reduce the number of predicates in the abstraction refinement process. In this

paper, we propose a different perspective to minimize number of predicates by analyzing counterexample. For any spurious counterexample, there is at least one variable that makes it infeasible. In contrast to BLAST and F-Soft where the computation of new predicates can be performed by using interpolation and weakest pre-conditions techniques, respectively, we traverse through the counterexample and use propagation-and-substitution of variables to find infeasible variables then assign new predicates. The proposed method produces equal or smaller number of predicates comparing to other approaches. The cost of traversing counterexample to find new predicates is small comparing to model checking and abstraction computation.

This paper is organized as follows. Some related researches on software model checking describes in next section. An illustrative example used to describe the idea of the proposed method is shown in Section 3. Formal definitions of predicate abstraction and refinement are presented in Section 4, while in Section 5 describes our approach of minimizing the number of predicates by analyzing the counterexample. Some experimental results comparing our approach with the public tool BLAST are presented in Section 6 and we conclude with Section 7.

2 Related Work

Software model checking poses its own challenges, as software tends to be less structured than hardware. In addition, concurrent software contains processes that execute asynchronously, and interleaving among these processes can cause a serious state-space explosion problem. Several techniques have been proposed to reduce the state-space explosion problem, such as partial-order reduction and abstraction. In the software verification domain, predicate abstraction [4,5,9] is widely applied to reduce the state-space by mapping an infinite state-space program to an abstract program of Boolean type while preserving the behaviors and control constructs of the original. Counterexample-Guided Abstraction Refinement (CEGAR) [10] is a method to automate the abstraction refinement process. More specifically, starting with a coarse level of abstraction, the given property is verified. A counterexample is given when the property does not hold. If this counterexample turns out to be spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process is continued until there is no error found or there is no solution for the given property. Toolkits of various platforms like C or the system-level design languages like SystemC or SpecC have been implemented. We briefly introduce some of them.

The SLAM project [5] conducted by Ball and Rajamani has developed a model checking tool based on the interprocedural dataflow analysis algorithm presented in [11] to decide the reachability status of a statement in a Boolean program. The generation of an abstract Boolean program is expensive because it requires many calls to a theorem prover.

Clarke et al. [12] use SAT-based predicate abstraction. During the abstraction phase, instead of using theorem provers, a SAT solver is used to generate the abstract transition relation. Many theorem prover calls can potentially be replaced

by a single SAT instance. Then, the abstract Boolean programs are verified with SMV. In contrast to SLAM, this work is able to handle bit-operations as well. This idea is also extended to use with SpecC language [13]. The synchronization constructs *notify/wait* can be modeled, but it does not explain how to handle the timing constraints that are introduced by using *waitfor*.

Sakunkonchak et al. [14] propose a SpecC/synchronization verification tool (S-VeT) based on the predicate abstraction of ANSI-C programs of SLAM project [5]. Concurrency and synchronization can be handled by mathematically model SpecC programs by equalities/inequalities formulae. These formulae are solved by using the Integer Linear Programming (ILP) solver.

3 Motivating Example

Similar to existing methods, abstraction and refinement processes are automated based on the state-of-the-art model checking and counterexample-guided abstraction refinement (CEGAR). More specifically, starting with a coarse level of abstraction, the given property (e.g. reachability of a statement that caused an error in the model) is used to verify the abstract model. A counterexample is given when this property is not satisfied. If this counterexample turns out to be spurious, the previous abstract programs are then refined to a finer level of abstraction. The verification process continues until there is no error found or there is no solution for the given property.

Let us consider the program as shown in Figure 1(a). We would like to check whether a statement labeled "ERROR" is reachable. Predicate abstraction of the original program is applied as shown in Figure 1(b) where *Abstract*0 abstracts *Original*. At this moment, we do not consider any particular predicate. Statements are not considered (represented as "...") and branching conditions are considered as non-deterministic choices (represented as "*"). Note that the behaviors and control constructs at every abstraction and refinement step are preserved.

When model check the abstract model *Abstract*0 in Figure 1(b), ERROR label is reachable as shown by an abstract counterexample which goes through all program locations starting from 1 to 8 (for simplicity, we omit the declaration of variables in line 0). Concretization of this counterexample with respect to *Original* is applied. *Concretize* as shown in the left of Figure 1(c) is obtained by corresponding every location from an abstract counterexample to the same location in *Original*. Decision procedure is used to simulate this concrete path. The concrete path *Concretize* is obviously *infeasible* (explain below). To find new predicates to refine *Abstract*0, the technique described in [8,7] can be used to remove the infeasible trace by tracking exactly one predicate at each program location from 1 to 8. The method in [7] can further remove more predicates by traversing the counterexample with weakest pre-conditions.

The proposed method is simpler and more efficient. Simulation of the concrete path and finding new predicates can be done by storing information while traversing the counterexample. *Concretize* is simulated starting from location 1.

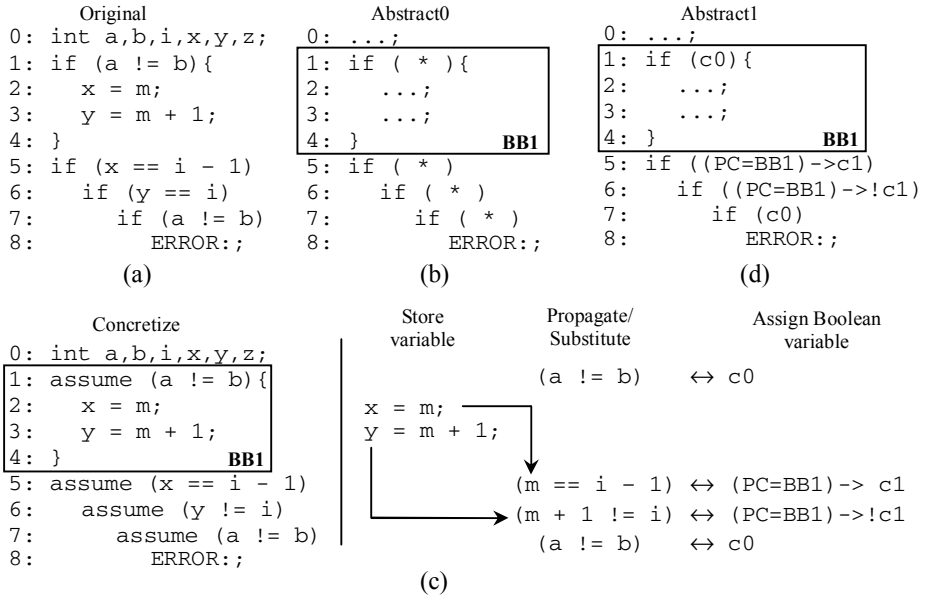

Fig. 1. (a) Original program. (b) First abstraction. The counterexample is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$. (c) Concretized the counterexample and use the propagate-and-substitute of variables x and y. Assign Boolean variables to corresponding predicates that caused infeasibility in counterexample. Note that only two predicates are used. (d) Abstract1 is the refined version of Abstract0 in (b). After model check, label ERROR is not reachable.

At each location during traversal, the program locations and assignment of variables are stored. For example, at location 2 and 3, the assignments of variables x and y are stored and these locations can be referred as they are in the same basic block $BB1$. As traversing to location 5 and 6, variables x and y are used. The symbolic values of x and y ($x \Rightarrow m$ and $y \Rightarrow m + 1$) in location 2 and 3 are propagating-and-substituting to location 5 ($x == i - 1$) and 6 ($y \mathrel{!}= i$). We obtain ($m == i - 1$) and ($m + 1 \mathrel{!}= i$) in location 5 and 6, respectively. This makes this counterexample infeasible. The process is repeated until error statement is found. Up to this point, the process of simulating the concrete path is the same with other methods except that the program locations and the assignments of variables are stored in memory. The concrete path is traversed and predicates that caused infeasibility are found. Boolean variables $c0, c1, !c1, c0$ can be assigned at location 1, 5, 6, and 7, respectively. However, at location 5 and 6, variables x and y depend on the execution of $BB1$. The conditions of dependent path must be added. Finally, the predicates assigned at location 1, 5, 6, 7 are $c0, (PC = BB1) \rightarrow c1, (PC = BB1) \rightarrow \ !c1, c0$, respectively.

Note that only two Boolean variables ($c0$ and $c1$) are used in this example. Although there is a program location ($PC = BB1$) to cooperate with $c1$, this is not an additional variable in the abstract model when performing model checking.

This is because all the program locations are already included in the abstract model. Therefore, only $c0$ and $c1$ are added for model checking and abstraction computation. According to this example, equal or larger number of predicates is produced by methods [8,7] and so as the cost of computation.

Finally, the abstract model *Abstract*1 in Figure 1(d) is constructed according to these new predicates. As the result of model checking, the error statement at location 8 is unreachable. Hence, this program is safe.

4 Predicate Abstraction and Refinement

4.1 Predicate Abstraction

Predicate abstraction is a technique to construct a conservative abstraction which maps an infinite state-space concrete program to an abstract program. Formally the concrete transition system is described by a set of *initial states* represented by $I(\bar{s})$ and a *transition relation* represented by $R(\bar{s}, \bar{s}')$ where S is a set of all states in concrete program and $\{\bar{s}, \bar{s}'\} \in S$ are a set of current and next states, respectively. The variables in the concrete program are represented by Boolean variables which correspond to one predicate on the variables in the concrete program. The abstraction is determined by a set of predicates $Pred = \{\phi_1, \ldots, \phi_k\}$ over the given program. If applying all predicates to a concrete state, a k-width vector of Boolean values, $\bar{b}$, is obtained. In other words, $\bar{b} = abs(\bar{s})$ maps the concrete states $\bar{s} \in S$ into the abstract states $\bar{b}$ where $abs(\cdot)$ is an *abstraction function.*

The term *conservative abstraction* means a transition from the abstract states $\bar{b}$ to $\bar{b}'$ in the abstract model exist if and only if there is a transition from $\bar{s}$ to $\bar{s}'$ in there concrete model where $\bar{b} = abs(\bar{s})$ and $\bar{b}' = abs(\bar{s}')$. The abstract initial states $\hat{I}(\bar{b})$ is

$$\hat{I}(\bar{b}) := \exists \bar{s} \in S : (abs(\bar{s}) = \bar{b}) \wedge I(\bar{s})$$

where the abstract transition relation $\hat{R}(\bar{b}, \bar{b}')$ can be shown as

$$\hat{R} := \{(\bar{b}, \bar{b}') | \exists \bar{s}, \bar{s}' \in S : R(\bar{s}, \bar{s}') \wedge (abs(\bar{s}) = \bar{b}) \wedge (abs(\bar{s}') = \bar{b}')\}$$

With this conservative abstraction, if the property $\hat{Prop}$ holds on an abstract state $\bar{b}$ in the abstract model, then the property must hold on all states $\bar{s}$ where $abs(\bar{s}) = \bar{b}$.

$$\hat{Prop}(\bar{b}) := \forall \bar{s} \in S : (abs(\bar{s}) = \bar{b}) \Rightarrow Prop(\bar{s})$$

Therefore, when model checking the abstract model, if property $\hat{Prop}$ holds on all reachable states, then *Prop* also holds on all reachable states in concrete model. Otherwise, an abstract counterexample is obtained. In order to check if this abstract counterexample corresponds to a concrete counterexample, we need to simulate this abstract trace by concretizing it with the concrete model. If the simulation result tells that the abstract trace is really not feasible in the concrete model, this abstract counterexample is then called the *spurious counterexample.* To remove the spurious behaviors from the abstract model, the refinement of abstraction is needed.

4.2 Abstraction Refinement

If the abstract counterexample cannot be simulated, it is because the abstraction is too coarse. Spurious transitions make the abstract model not corresponding to the concrete model. In order to eliminate the spurious transitions from the abstract model, we find variables that caused infeasibility in the concrete trace.

A sequence of length $L+1$ in the abstract model is a sequence of abstract states, $\bar{b}_0, \ldots, \bar{b}_L$ such that $\hat{I}(\bar{b}_0)$ holds and for each i from 0 to $L-1$, $\hat{R}(\bar{b}_i, \bar{b}_{i+1})$ holds. An *abstract counterexample* is a sequence $\bar{b}_0, \ldots, \bar{b}_L$ for which $\hat{Prop}(\bar{b}_L)$ holds. This abstract trace is *concrete* or so called *real counterexample* if there exists a concrete trace corresponding to it. Otherwise, if there are no concrete traces corresponding to this abstract trace, then it is called a *spurious counterexample*.

The abstraction function was defined to map sets of concrete states to sets of abstract states. The *concretization function*, $conc(\cdot)$, does the reverse. There is a concrete counterexample trace $\bar{s}_0, \ldots, \bar{s}_L$, where $\bar{s}_i$ corresponds to a valuation of all the k predicates $\phi_1, \ldots, \phi_k$, corresponding to the abstract counterexample trace $\bar{b}_0, \ldots, \bar{b}_L$ if these conditions are satisfied:

- For each $i \in 0, \ldots, L$, $conc(\bar{b}) = \bar{s}$ holds. This means that each concrete state $\bar{s}_i$ corresponds to the abstract state $\bar{b}_i$ in the abstract trace.
- $I(\bar{s}_0) \wedge R(\bar{s}_i, \bar{s}_{i+1}) \wedge \neg Prop(\bar{s}_L)$ holds, for each $i \in 0, \ldots, L-1$. With this, the concrete counterexample of length $L+1$, starting from initial state, exists.

Thus, with the following formula,

$$\bigwedge_{i=0}^{L-1} \left(\bigwedge_{j=1}^{k} \phi_j = \bar{s}_i \right) \wedge \bigwedge_{i=0}^{L-1} R(\bar{s}_i, \bar{s}_{i+1})$$

if it is satisfiable then the abstract counterexample is concrete. Otherwise, it is a spurious counterexample.

5 Spurious Counterexample Analysis for Predicate Refinement

Let $|P|$ be the size of a given program (number of statements) and $|Pred|$ be the number of predicates in the abstraction refinement process. Computation cost of this model is $|P| \cdot 2^{|Pred|}$. It is obvious that the smaller the number of predicates in the abstraction refinement process, the exponential reduction in the cost of abstraction computation and model checking. Also, the size of the program can be considered.

This section describes the proposed method to reduce the size of the abstract model used in abstraction refinement process. The program size $|P_{BB}|$ where $|P_{BB}| \leq |P|$ is considered (instead of number of statements, number of basic blocks are considered). The number of predicates used in the proposed method

is $|Pred_{Proposed}|$ where $|Pred_{Proposed}| \leq |Pred|$. Thus, the computation cost of the proposed method is

$$|P_{BB}| \cdot 2^{|Pred_{Proposed}|} \leq |P| \cdot 2^{|Pred|}$$

Given a spurious counterexample, the proposed abstraction refinement method by analyzing the spurious trace is shown in Algorithm 1. In [7], localization and register sharing methods are used to reduce the number of predicates used in refinement process. Their refinement algorithm performs a backward weakest precondition propagation for each branching condition (`assume` statement) in the infeasible trace.

The abstraction refinement algorithm described in Algorithm 1 performs an analysis by traversing the spurious counterexample from the initial state. First, the spurious trace is traversed and information on the program locations and the assignments of variables are stored. Then, it is traversed again to find the conflict variables which can be represented by a conflict predicate. If this predicate is depending on execution of some program locations, then those program locations are associated to the predicate in the abstract model.

In [7], the number of predicates used depends proportionally on the number of *branching conditions*. In contrast, the number of predicates used in our approach is depending on the number of *conflict predicates*. Back to the example shown in Section 3, the method of [7] needs *four* predicates *plus* extra global constraints to make the relationship of those predicates more precise, while the proposed method needs *only two* predicates with some path dependences. And these conditions of path dependences do not make the abstract model larger because they are already represented the abstract states in the abstract model.

6 Experimental Results

The proposed method for abstraction refinement by counterexample analysis is implemented in SpecC/Synchronization Verification Tool (S-VeT) [14]. S-VeT tool accepts SpecC descriptions as input. In order to compare with BLAST, we generate SpecC wrappers for the C descriptions that used to verify with BLAST. These wrappers just make the program to be able to process by S-VeT tool. They do not introduce additional behaviors to the program.

The experiments are performed on a Pentium4 2.8GHz machine with 2GB memory running Linux. Several experiments are conducted to compare our technique against a public C-based verification tool, BLAST.

As reported in [7], BLAST fails to find the new set of predicates during refinement when applying with the default predicate discovery scheme. Options `craig2` and `predH7` were reported to give best performance. For comparison in this paper, we use the same options. Table 1 shows the comparison of various benchmarks running BLAST against our approach. Size of the program can be defined by the number of branching conditions used. The "TP" column gives the to total number of predicates in the program. "Pred" columns give the maximum number of predicates active at any program location in refinement process. Runtime related columns, "Abs", "MC", "Ref" and "Time" denote the execution

Algorithm 1. Abstraction refinement by analyzing counterexample to find conflict predicates and program location dependents

Declare

A spurious counterexample $\bar{s}_0, \ldots, \bar{s}_L$
$Pred$ is a set of predicates (for simplicity, the subscript $_{Proposed}$ is omitted)
ϕ is a predicate that found to make counterexample infeasible
Program locations are defined in term of basic block instead of individual statements (to make abstract model smaller)
The stored information $Store = (PC, Var, Symb)$ where PC is a set of program locations, Var is a set of variables to be assigned, and $Symb$ is a set of symbolic values corresponding to the variable in that location

Begin

/* Initially, store program locations and variable assignments */
for $i = 0$ to L **do**
    if $\bar{s}_i$ is an assignment **then**
        PC_i := program location
        Var_i := the assigned variable (left hand side of the assignment)
        $Symb_i$:= symbolic value (right hand side of the assignment)
    end if
end for

/* Traverse counterexample to find conflict variables and assign new predicates */

for $i = 0$ to L **do** /* Outer loop */
    if $\bar{s}_i$ is a branching condition **then**
        for $j = 0$ to i **do** /* Inner loop #1 */
            if variables in $\bar{s}_j$ contains the variable Var_j **then**
                Propagate-and-substitute all existence of Var_j in $\bar{s}_j$
            end if
        end for
        Simulate this path $\bar{s}_0, \ldots, \bar{s}_j$
        if predicate ϕ presents a conflict in the path is found **then**
            for $j = i$ downto 0 **do** /* Inner loop #2 */
                Track back to find if variables in ϕ are depending on any program location PC_j
                if variables in ϕ depends on execution of PC_j **then**
                    Associate PC_j to ϕ when constructing abstract model[1]
                end if
            end for
            $Pred := Pred \cup \phi$
        end if
    end if
end for

End

[1]Abstract model $M = (PC, \phi)$ and we can check M using NuSMV.

Table 1. Experimental results comparing our approach implemented in S-VeT againt BLAST verification tool

Benchmark	TP	BLAST		S-VeT					Bug
		Pred	Time	Pred	Abs	MC	Ref	Time	
TCAS0	54	13	20.97	12	5	9	10	24	No
TCAS1	111	37	381.20	20	24	58	39	**122**	No
TCAS2	55	13	22.38	13	3	10	10	23	No
TCAS3	63	14	25.84	13	5	12	9	26	Yes
TCAS4	83	23	41.56	13	7	18	13	**39**	Yes
TCAS5	73	20	39.78	15	6	15	7	**28**	No
TCAS6	73	19	32.82	10	3	12	10	**25**	Yes
TCAS7	74	18	33.18	16	4	16	15	36	Yes
TCAS8	61	14	31.99	11	7	11	9	**28**	No
TCAS9	83	20	55.78	14	8	22	17	**38**	Yes
PredAbs1	29	26	1.38	1	0.45	0.97	0.99	2.45	No
PredAbs2	57	52	21.88	1	0.43	1.54	1.46	**3.44**	No
PredAbs3	88	78	140.16	1	0.43	3.46	2.28	**6.19**	No

time in seconds for abstraction, model checking, refinement process, and total runtime, respectively. The pre-process time is omitted.

The benchmarks Traffic Alert and Collision Avoidance System (TCAS) are tested with ten different properties. The benchmarks PredAbs are the fabricated examples used to validate the predicate abstraction refinement process. In all PredAbs benchmarks, only one predicate is sufficient to show that the counterexample is spurious. While our approach can find this predicate directly from analyzing the spurious trace, BLAST needs to interpret a large set of predicates before it knows that this trace is spurious. Although, BLAST is implemented with lazy abstraction technique and the complex refinement scheme based on Craig interpolation method, our approach can outperform by 8 out of the total of 13 benchmarks.

7 Conclusion

Predicate abstraction is a common and efficient technique in software verification domain. However, when the size of the program (number of branching conditions) is large, predicate abstraction suffers from the computation cost that increases exponentially as the number of predicates increases. Choice of predicate selection, scope and number of predicates are major artifacts for performance improvement. In this paper, we described a technique for improving the performance of the abstract refinement loop by analyzing the spurious counterexample. In order to get the smaller set of predicates, we analyze the spurious counterexample to find the conflict predicates. If any predicate is depended on any program location, that program location was associated with that predicate in the abstract model. Our approach is implemented with S-VeT toolkit. Experimental results present the

comparison of our method against BLAST toolkit. The results show that fewer number of predicates can be found with our approach.

Acknowledgement

We would like to thank Himanshu Jain for the Traffic Alert and Collision Avoidance System (TCAS) benchmarks.

References

1. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
2. Clarke, E.M., Grumberg, O., Dill, D.E.: Model checking and abstraction. In: ACM Transactions on Programming Languages and System TOPLAS, vol. 16 (1994)
3. Godefroid, P.: Model checking for programming languages using verisoft. In: Proc. of the 24th ACM Symposium on Principles of Programming Languages (1997)
4. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, Springer, Heidelberg (1997)
5. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, Springer, Heidelberg (2001)
6. Henzinger, T.A., Jhala, R., Mujumdar, R., Sutre, G.: Lazy abstraction. In: ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (2002)
7. Jain, H., Ivancic, F., Gupta, A., Ganai, M.K.: Localization and register sharing for predicate abstraction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
8. Henzinger, T.A., Jhala, R., Mujumdar, R., McMillan, K.L.: Abstractions from proofs. In (POPL 2004). Proc. of the 31st Annual Symposium on Principles of Programming Languages (2004)
9. Henzinger, T.A., Jhala, R., Mujumdar, R., Sutre, G.: Lazy abstraction. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, Springer, Heidelberg (2003)
10. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
11. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In (POPL 1995). Principles of Programming Languages (1995)
12. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. In: Proceeding of the Model Checking for Dependable Software-Intensive Systems Workshop (2003)
13. Clarke, E.M., Jain, H., Kroening, D.: Verification of SpecC using predicate abstraction. In: Second ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE 2004) (2004)
14. Sakunkonchak, T., Komatsu, S., Fujita, M.: Synchronization verification in system-level design with ILP solvers. In: Third ACM-IEEE International Confernece on Formal Methods and Models for Codesign (MEMOCODE2005) (2005)