AUTOABS: Syntax-Directed Program Abstraction

Nina Amla[1], Robert P. Kurshan[1], and Kedar S. Namjoshi[2]

[1] Cadence Design Systems
{namla,k}@research.bell-labs.com
[2] Bell Laboratories, Lucent Technologies
kedar@research.bell-labs.com

Abstract. We present the AUTOABS tool, which carries out automatic, syntax-directed program abstraction, prior to model checking. The heart of the tool is an algorithm for predicate abstraction from [NK00]. We briefly describe this algorithm, then discuss implementation issues, the analysis of error traces, and directions for current and future work.

Keywords: Abstraction, Model Checking, Static Program Analysis

Corresponding Author: Kedar S. Namjoshi
Address: 2C-301, 600-700 Mountain Ave., Murray Hill, NJ 07974, U.S.A.
E-mail: kedar@research.bell-labs.com
Phone: +1 (908) 582-1891

This page left blank.

1 Introduction

We present the AUTOABS tool, which carries out automatic, syntax-directed abstraction of S/R programs[1] relative to a property, prior to model checking. While model checking [CE81,QS82] is a successful verification method, and is used widely in industry, it is still the case that only programs with fairly small state spaces can be verified fully automatically: for larger programs, one has to apply guided decomposition and abstraction methods. In predicate-based abstraction methods, the data in a program is abstracted by only keeping track of certain predicates (constraints) on the data; each predicate is represented by a boolean variable in the abstract program, while the original data variables are eliminated.

Early applications of this method (for example, [GS97]) were dependent on the user identifying the set of predicates that influence the correctness property. For the method to scale to larger programs, however, it is essential to have an *automated* algorithm to compute these predicates. The algorithm should not, of course, exhaustively explore the global state space of the program, since this would defeat the purpose of abstraction. AUTOABS implements the algorithm described in [NK00], which computes relevant predicates and an abstract program based purely on a syntax-directed analysis of the program *text*. The analysis is based on weakest preconditions [Dij75] (*wlp*'s), which can be calculated by syntactic substitution. It also enjoys a completeness property: if the concrete program has an equivalent finite program, relative to bisimulation or simulation on the predicates in the property, the algorithm can compute a (possibly, different) finite and equivalent abstract program. It is these two properties, together with the simplicity of the *wlp* computation, which make it a promising candidate for implementation. In this paper, we briefly describe the algorithm, then discuss implementation issues, the analysis of error traces, and directions for current and future work.

2 Example: A FIFO Queue

We illustrate the abstraction algorithm with a simple example of a FIFO queue. The S/R program for the queue is given in the left half of Figure 1. State variables are called `stvar`'s. The value of a `stvar` in the next state is given by the expression to the right of the $\rightarrow$ in its `asgn` (assignment) statement. Combinational (0-delay) variables are called `selvar`'s, and their values are given by the expressions to the right of the $:=$, where $\{*\}$ denotes non-deterministic choice (i.e., an input). Suppose that we want to prove that whenever the input value is 0, subsequently, the output value is 0 – in temporal logic, $\mathsf{G}((Q.in = 0) \Rightarrow \mathsf{XF}(Q.out = 0))$. In a direct check of this property without data abstractions, COSPAN runs out of memory for queue depths greater than 32 (i.e., $N > 32$). On the other hand, one could reason as follows for $N = 3$: $Q.out = 0$ only if $Q.x[2] = 0$, which holds only if $Q.x[1] = 0$ at the previous step, which holds only if $Q.x[0] = 0$ two steps

[1] S/R is the input language of the COSPAN model checker [HHK96].

```
/* N=queue depth */
#define N 3
#define Data (0..31)
proc Q
 selvar in,out:Data
 asgn   in := {*},
        out := x[N-1]

 stvar x[N]:Data
 init x[0..N-1] := 0
 asgn x[0] -> in
 asgn [i in 0 .. N-2]
       {x[i+1] -> x[i]}
end Q
```

```
proc Abstract_Q
/* b0:(Q.out=0), b4:(Q.in=0) */
 selvar b0,b4:boolean
 asgn b4 := {*},
      b0 := b1

 /* b1:(Q.x[2]=0), b2:(Q.x[1]=0),
    b3:(Q.x[0]=0) */
 stvar b1,b2,b3:boolean
 init b1:=true,b2:=true,b3:=true
 asgn b1 -> b2,
      b2 -> b3,
      b3 -> b4
end Abstract_Q
```

Fig. 1. The FIFO Queue and its abstracted form

earlier, which holds only if $Q.in = 0$. Thus, the property really depends only on the boolean values of these five predicates.

The *wlp* computations in the algorithm from [NK00] *simultaneously* determine the relevant predicates, their mutual dependencies, and their updates. The algorithm creates a boolean variable in the abstract program for each new predicate that is encountered. For instance, $wlp(Q.x[2] = 0)$ is calculated by substituting $Q.x[2]$ with its assignment expression, obtaining $(Q.x[1] = 0)$. As $b1$ is the boolean variable for $(Q.x[2] = 0)$, a new variable $b2$ is created for $(Q.x[1] = 0)$, and the update to $b1$ is computed as "`asgn b1 -> b2`". Proceeding in this manner from $b0 : (Q.out = 0)$, the abstract program computed by AUTOABS is shown in the right half of Figure 1 (slightly re-formatted for readability). For a queue of depth N, all relevant predicates are computed with $N + 1$ *wlp* calculations, resulting in $N + 2$ boolean variables in the abstract program. For the case of $N = 32$, AUTOABS takes 4 seconds to calculate the abstract program, and less than a second to verify the property!

For this example, the predicates stay the same even if the data domain is infinite; for instance, the set of integers. In general, for computability reasons, it is not possible for this abstraction process to always terminate, so we may need to limit the number of *wlp* iterations – details are in [NK00].

3 Implementation

The structure of the tool is given in Figure 2. It consists of four main blocks, whose functionality is described below.

S/R parser: The parser takes an S/R program and a property automaton, also specified in S/R, and generates a parse tree of the composition, flattening out both program hierarchy and array structure. The resulting program consists of scalar variables, initializations, and assignment statements. The initializations are all constants, and define a single initial state. Assignments are rewritten

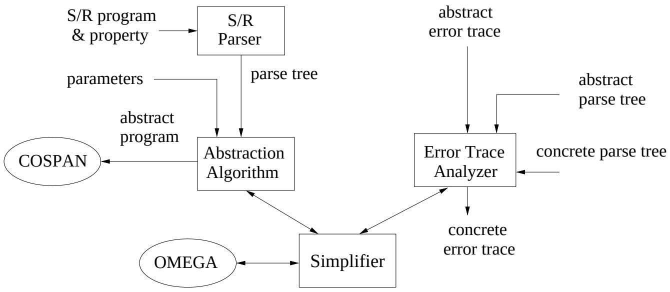

Fig. 2. Organization of AUTOABS

to three types: non-deterministic assignments to input variables; deterministic, acyclic assignments to combinational variables; and deterministic assignments to state variables. The determinism is crucial to the algorithm, since it permits the computation of *wlp* by syntactic substitution.

Abstraction: This block implements the predicate abstraction algorithm from [NK00]. As described earlier, the algorithm consists of iterated passes of *wlp* calculations followed by simplification. The set of relevant predicates is initialized to those that appear in the acceptance condition for the property automaton. Un-abstracted control variables and their assignment statements are copied over to the resulting (abstract) S/R program. The algorithm has two parameters: the maximum number of iterations, and a lower bound on the size of domains that should be abstracted. All variables with domains of size greater than this bound are candidates for abstraction.

Simplifier: This block consists of a simplifier that computes constant expressions, simplifies if-then-else, boolean, and arithmetic operators, does early quantification (e.g., distributing $\exists$ over $\vee$), and handles difficult Presburger arithmetic constraints using the Omega simplifier [Pug92].

Error Trace Analyzer: Since the tool produces *conservative* abstractions, if the abstract program satisfies the property, so does the original. Error traces of the abstract program, however, may not always be *feasible*, i.e., represent concrete error traces. This is because the abstract program is, generally, an over-approximation. Moreover, abstract error traces are given in terms of boolean values for each predicate and are, therefore, hard to analyze. The error trace analyzer solves these problems by checking an abstract error trace for feasibility, and producing a concrete error trace if the abstract trace is feasible. This is done by a backward analysis that builds up a formula describing those concrete states from which there is an execution that matches (in terms of predicate values) the values in the abstract trace. Consider an abstract error trace $a_0, \ldots, a_{n-1}$ for a safety property. Let $\hat{a}_i$ be the set of concrete states represented by the predicate values in a_i (e.g., for $c_0 : (x > 0), c_1 : (x = 10)$ and $a_i = (c_0, \neg c_1)$, $\hat{a}_i \equiv (x > 0) \wedge \neg(x = 10)$). The formula is built up by the recurrence $C_{n-1} =$

$\hat{a}_{n-1}$, $C_i = \hat{a}_i \wedge \mathsf{EX}(C_{i+1})$. The concrete initial state is substituted into C_0: the trace is feasible if, and only if, C_0 evaluates to *true*.

The EX computation is the following: $[\mathsf{EX}(P)](x) \equiv (\exists i, x' : T_C(x, i, x') \wedge P(x'))$, where P is boolean formula on concrete program states, and T_C is the transition relation of the concrete program. This computation is performed, as far as possible, by syntactic substitution, using the determinism of the assignment statements – quantifier elimination is needed only for abstracted data input variables (e.g., $Q.in$). The analysis algorithm sketched above has been extended to apply to error traces for liveness properties, and to search for matching concrete traces that can be longer than the abstract trace. On the queue example, with $N = 32$ and the (false) safety property $\mathsf{G}(Q.out = 0)$, the error trace analysis and generation of a matching concrete trace takes about 10 seconds.

The current implementation of the tool consists of about 7000 lines of OCaml code, distributed as follows: parser (2900), simplifier (2500), the abstraction algorithm (only 1000), and error trace analysis (500). Implementation with the OCaml environment has been an enjoyable experience.

4 Related Work and Conclusions

Many model checking tools perform data abstraction by restricting the domain of data variables; examples include COSPAN, Cadence SMV [McM98], and Murϕ [ID96]. Such heuristics are successful for data-independent programs, but are usually inapplicable to programs that include comparisons and arithmetic, for which methods based on predicate abstraction are better suited. Implementations of predicate abstraction with ALDÉBARAN and PVS [GS97,SS99] leave the discovery of relevant predicates to the user, which makes them less effective for large programs.

Recently, tools have been developed where predicate discovery is guided by analysis of error traces [CGJ$^+$00,BR01]. In [CGJ$^+$00], the algorithm (a refinement of localization reduction [Kur94]), is based on BDD representations of the program. This is a drawback for analyzing large programs, where transition relation BDD's are often too large for efficient manipulation. The algorithm from [BR01] does not use BDD's, but is restricted to safety properties. Both of these algorithms may be summarized as the process *[(1-step abstraction; model checking; error-trace analysis)*]*. Our algorithm, in contrast, can be summarized as *[(1-step abstraction)*; model checking]*. Repeated syntax-directed abstraction is cheaper than repeated model checking; therefore, we expect our algorithm to scale better for larger programs. Error-trace analysis is not essential for completeness, which can be achieved by just increasing the number of *wlp* iterations, but it is useful later, in the analysis of abstract error traces. Our error-trace analysis algorithm appears to be new; the trace analysis algorithms in [CGJ$^+$00,BR01] use forward computation from the initial state. Computing backwards using EX allows us to take advantage of syntactic substitution.

We have experimented with AUTOABS on a number of small- and medium-size S/R programs, with good results. Our current goals include the addition of array-slice processing and an interface to Cadence's FormalCheck tool, which will allow the direct analysis of larger programs written in VHDL and Verilog.

References

[BR01] T. Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*. Springer Verlag, 2001.

[CE81] E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*. Springer Verlag, 2000.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *CACM*, 18(8), 1975.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, 1997.

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102 of *LNCS*, 1996.

[ID96] C.N. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2), 1996.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV*, volume 1427 of *LNCS*, 1998.

[NK00] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV*, volume 1855 of *LNCS*. Springer Verlag, 2000.

[Pug92] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *CACM*, 35(8), 1992.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, volume 137 of *LNCS*, 1982.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV*, volume 1633 of *LNCS*. Springer Verlag, 1999.