Automatic Generation of Invariants in SPIN

Mandana Vaziri* and Gerard Holzmann†

August 19, 1998

Abstract

SPIN takes a model to be verified and a property, and outputs true if the model satisfies the property or otherwise false, with a counterexample. In the first case, the user has no indication about why the property is satisfied. It may be the case that the model does not really represent what the user meant to express, and the property is void. It can therefore be useful if the tool could provide some additional information about the behavior of a model, regardless of a property's validity.

A way of providing such information is to have SPIN discover invariants of the model automatically. In this paper we focus on generating invariants of the form *a rel b*, where *a* and *b* are integer variables, and *rel* is an ordering relation. We give an algorithm with a matching implementation in SPIN, including a graphical user interface.

1 Introduction

A model checker takes a model and a property as inputs and determines whether the model satisfies the property. Its output is either true, if the model satisfies the property, or false. In the latter case, the user has an indication of why the property is not satisfied in the form of a counterexample. But if the output is true, the user has no indication why the property is true. In this case, it could be that the property is void, or that the model is invalid. It would be therefore useful for the model checker to output some information about the behavior of the model even if the property is satisfied.

One way of conveying such information is to output invariants of the model automatically. There are several existing methods for generating invariants. *Forward propagation* generates invariants by executing the model forward, either starting from the start state [3] or from some intermediate state [4]. In the former work, the invariant obtained is an assertion that characterizes the set of reachable states of a model. In the latter work, *local invariants* - invariants that are true of a particular control location - are propagated to other control locations. *Backward propagation*

*vaziri@theory.lcs.mit.edu

†gerard@research.bell-labs.com

allows for the strengthening of an existing invariant [4, 3] by calculating the weakest precondition with respect to that invariant and taking it as a conjunct. Finally, local invariants of components can be combined into global invariants of a composition [4].

Invariants generated by the methods above quickly become complicated. For example, assertions that characterize the set of reachable states may be very large. Even though, these invariants may be appropriate for use by an automated tool such as a theorem prover, they may not convey useful conceptual information to a human user.

Our approach is to consider different forms of invariants one at a time and to focus on generating that form. In this approach, the invariants are simpler and more focused on a particular kind of information. These invariants may still be used by a theorem prover but they are primarily intended to be interpreted by the user of the model checker directly.

We start by generating invariants of the form a *rel* b, where a and b are integer variables and *rel* is an ordering relation. We give an implementation of this algorithm in the model checker SPIN [2], and provide a matching graphical user interface.

The outline of the paper is as follows. Section 2 presents the algorithm and section 3 the implementation. Section 4 gives some examples. Finally, section 5 presents discussion and future work.

2 Invariant Generation Algorithm

In this section we describe the algorithm for the generation of invariants of the form a *rel* b, where a and b are integer variables and *rel* is an ordering relation.

2.1 Informal Description

The algorithm must be coupled with a state searching algorithm, which can be either be depth-first or breadth-first, but that guarantees that all the reachable states of the model are visited.

At each reachable state of the model, the algorithm determines the ordering relations for each pair of integer variables, and records them in a matrix of all invariant relations seen so far. After examining all reachable states, the algorithm outputs all global invariant relations discovered.

2.2 Notation

We use $\mathcal{V}$ to denote the set of integer variables of the model. We use v to denote an element of $\mathcal{V}$. We use $\mathcal{R}$ to denote the set $\{\perp, =, <, >, \leq, \geq, \not\perp\}$, and r an element of $\mathcal{R}$.

Let $\mathcal{M}(v_1, v_2)$ be a function that takes two integer variables and returns an element of $\mathcal{R}$.

Let $\mathcal{C}(v_1,v_2)$ be a function that takes two variables and returns their ordering relation. This function will be used to obtain the relation between variables in the

```
seen := empty
foreach v_1, v_2, M(v_1,v_2) := ⊥

inv-update(v_in, i_in) {
    if (v_in ∉ seen) then seen := seen ∪ {v_in}
    foreach v ∈ seen except v_in {
        Let v_1, v_2 such that v_1 = min_lex(v, v_in) and v_2 = max_lex(v, v_in)
        M(v_1,v_2) := combine(C(v_1,v_2), M(v_1,v_2))
}

inv-verdict() {
    foreach v_1,v_2 ∈ seen such that M(v_1,v_2) ∉ {≁, ⊥}
        print v_1 M(v_1,v_2) v_2
}

combine(r_new, r_old) {
    if r_new = r_old then return r_old
    else if {r_new, r_old} ⊂ {=, <, ≤} then return ≤
    else if {r_new, r_old} ⊂ {=, >, ≥} then return ≥
    else return ≁
}
```

Figure 1: Pseudo-code for Invariant Generation

current state.

2.3 Algorithm

The external interface to the invariant generation algorithm contains the following routines.

- *inv-update*(v, i): is called by the state searching algorithm whenever there is an assignment to variable v with value i.
- *inv-verdict*$()$: is called by the state searching algorithm when all the reachable states have been visited. This routine gives the invariants discovered.

The pseudo-code for invariant generation is shown in Figure 1. The variable *seen* is a set of integer variables that have already been seen by the algorithm, and is initially *empty*. $\mathcal{M}$ holds the invariant relations seen so far between different variables, and is initially set to all $\perp$.

inv-update(v_{in}, i_{in}) works as follows. If v_{in} has not been seen yet then it is added to the *seen* set. Then for each v in *seen* except v_{in} itself, the function $\mathcal{M}$ is updated at (v_1,v_2), where v_1 is the smallest of v and v_{in} lexicographically and v_2 the largest.

$\mathcal{M}$ is a triangular matrix representing the ordering relations between variables, and it is updated using the function *combine*.

We pass to *combine* the new ordering relation between v_1 and v_2 as given by the function $\mathcal{C}$, and the current invariant relation between them as stored in $\mathcal{M}$. The routine works as follows. If the new relation between v_1 and v_2 is the same as before, then that relation is returned. If the new and old relations are different but can be combined into $\leq$ ($\geq$) then $\leq$ ($\geq$) is returned. Otherwise, the function returns $\not<$ which indicates that no invariant ordering relation exists between v_1 and v_2.

inv-verdict is called when all reachable states have been attained. It simply prints out the invariant relations, if any.

The extra memory, in addition to what is needed for state searching, is $O(n^2)$, where n is the number of integer variables.

3 Implementation

This section describes an implementation of the invariant generation algorithm described above within the context of SPIN. Our implementation is linked to the depth-first search algorithm employed by SPIN to explore the state space.

There are a few abstract concepts in the algorithm that need to be implemented. First, we need to define what variables can be compared in the context of SPIN. For example, local variables of PROMELA processes cease to exist when the processes terminate. The functions $\mathcal{M}$ and $\mathcal{C}$ must also be developed. The following subsections explain how we implement these concepts. We conclude with a description of the graphical user interface.

3.1 PROMELA Variables

PROMELA has global variables and local variables belonging to process instances. Local variables are created and destroyed during the execution of a program along with the processes to which they belong. A local variable is uniquely characterized by its name, its *proctype* which is the integer corresponding to its process type, and its *pid* which is the integer corresponding to its process instance. Thus PROMELA variables can be thought of as triples of the form *(name, proctype, pid)*. Global variables may also be viewed in this way by giving special values to proctype and pid.

An *abstract variable*, corresponding to the variables used in the description of the algorithm above, corresponds to a set of PROMELA variables having the same name and proctype belonging to existing process instances. An abstract variable *(name, proctype)* may represent an empty set, in which case there are no existing process instances of type *proctype* having the variable *name*. We use the notation $\bar{v}$ to denote a PROMELA variable represented by abstract variable v.

We define the ordering relation between two abstract variables as follows. Let v_1 and v_2 be two abstract variables and $r \in \mathcal{R}$,

- If v_1 is empty or v_2 is empty or $\forall_{\bar{v_1},\bar{v_2}}$, $\bar{v_1}\ r\ \bar{v_2}$, where $r \in \{=,<,>,\leq,\geq\}$, then $v_1\ r\ v_2$.

- Otherwise, $v_1 \not\sim v_2$.

We say that $v_1 \; r \; v_2$ is an *invariant* if the statement is true of all reachable states of the program. We use the shortcut notation $\bar{v} \; r \; v_1$, to denote $\{\bar{v}\} \; r \; v_1$.

To reflect the fact that we have to deal with PROMELA variables, we modify our interface routines. These are given in C as follows.

- `void inv_update(char *a, int value, long int proctype, long int pid)`
- `void verdict()`
- `void inv_start(long int proctype, int pid)`
- `void inv_stop(long int proctype, int pid)`

The last two routines are used by SPIN to tell our implementation of the creation and termination of processes.

The invariant generation implementation produces relations of abstract variables as described above.

3.2 Implementing $\mathcal{M}$

The main data structure used in the implementation is a two-dimensional triangular matrix implemented with linked lists. An entry in the matrix corresponds to the ordering relation between two abstract variables. This ordering is represented by an integer (0: $=$, 1: $<$, 2: $>$, 3: $\leq$, 4: $\geq$). We also use -1 to denote the ordering relation $\not\sim$ described in the previously. The initial relation $\perp$ is represented by the lack of an entry in the matrix.

3.3 Implementing $\mathcal{C}$

We implement the function $\mathcal{C}$ with a C routine named `comp`, which gives the new relation between the abstract variables v_1 and v_2 in the current state. To this end, this function needs to access the values of the variables represented by v_1 and v_2. One way is to have the implementation access data structures maintained by the SPIN code. However, this is difficult in practice, because our implementation sees variables as `char *` and these cannot be used to access fields of C structures very easily.

Instead we keep an image of the current state in our implementation. Thus each time an update occurs, the new value of a variable is stored in a data structure named `pidList`. This data structure does not take a lot of memory since it only stores one state at a time.

To find the relation between v_1 and v_2, `comp` needs to compare each PROMELA variable in v_1 to each PROMELA variable in v_2. Then, the relation returned is combined with the current invariant relation stored in $\mathcal{M}$. Since only one PROMELA variable is updated at a time, say $\bar{v_1}$ without loss of generality, `comp` only needs to compare $\bar{v_1}$ with all the variables represented by v_2. Thus in the implementation, `comp` takes the value that was updated and an abstract variable, compares the PROMELA variables one by one and combines the result using a similar routine to *combine* described above.

3.4 Invoking the routines from SPIN

The new routines add a modest amount of overhead to model checking runs, and they are therefore only enabled when the user explicitly requests the computation of the invariants. To enable them, the user compiles the model checking code generated by SPIN with an extra compile-time directive -DINVARIANT. This will cause the invariant tracking code to be compiled in, and invoked when the model checking run is performed. At the end of the run the results can be displayed in either textual or in graphical form, as illustrated in the next section.

3.5 Graphical User Interface

The invariants can be communicated to the user via a graphical user interface. The relations are then represented as a graph, where each node corresponds to a variable and each edge to an ordering relation.

The interface can provide different views of the invariants, selecting invariants either by type (1 bit, 8 bits, 16 bits, etc.) or by name. In the latter case, for instance, only relations that correspond to a specific, named, variable are represented.

We modified the interface routines such that they take the type of the variable being updated as well. The type information is used by the graphical interface to provide different views by type.

4 Examples

This section presents a simple example of invariant generation with the extended version of SPIN, using a model from the standard SPIN distribution.

4.1 Peterson's Mutual Exclusion

The PROMELA code for the 2 process Peterson's algorithm is shown in Figure 2. We use this example to illustrate how adding constants to a model can cause the invariant generation implementation to output interesting invariants. In this case, we add the constant 1 (`tester` constant), with the goal of obtaining the property that at most one process must be in its critical section at a time.

The protocol has three global variables, `turn`, `flag`, and `ncrit`. The `turn` and `flag` variables are used to ensure mutual exclusion. `ncrit` is incremented each time a process enters its critical section. The global constant `tester` is included for comparison of other variables to 1.

Figures 3, 4, and 5 show the output of the invariant generation program. Each node corresponds to a variable of the model, and the edges represent ordering rela-

```
bool turn, flag[2];
byte ncrit;
byte tester = 1;

active [2] proctype user()
{
assert(_pid == 0 || _pid == 1);
again:
flag[_pid] = 1;
turn = _pid;
(flag[1 - _pid] == 0 || turn == 1 - _pid);

ncrit++;
assert(ncrit == 1); /* critical section */
ncrit--;

flag[_pid] = 0;
goto again
}
```

Figure 2: 2-process Peterson's Algorithm in PROMELA

tions. The edges are labeled with the order they represent, in this case greater than or equal to. The output shows that the constant 1 is greater than or equal to all the other variables.

The view menu provides a series of commands for modifying the presentation of the graph. Figure 4 shows choosing the view by variable `ncrit`. Then Figure 5 presents the outcome of this command. Only the variables relating to `ncrit` are shown. Since `ncrit` is the number of processes that are in their critical sections, this graph presents the mutual exclusion property.

5 Discussion and Future Work

The invariant generation algorithm was tested on the suite of examples distributed with SPIN. These experiments revealed small inefficiencies in some of the models, such as variables that are never assigned to or variables that have too generous a type assigned to it. Other observations we have made are:

- Invariants give feedback about the model even if the verification runs out of memory.

- Dummy variables with constant values can be added to the model to make the system output properties of interest such as the mutual exclusion property presented above. In addition, constants may be useful for determining upper or lower bounds for variables.

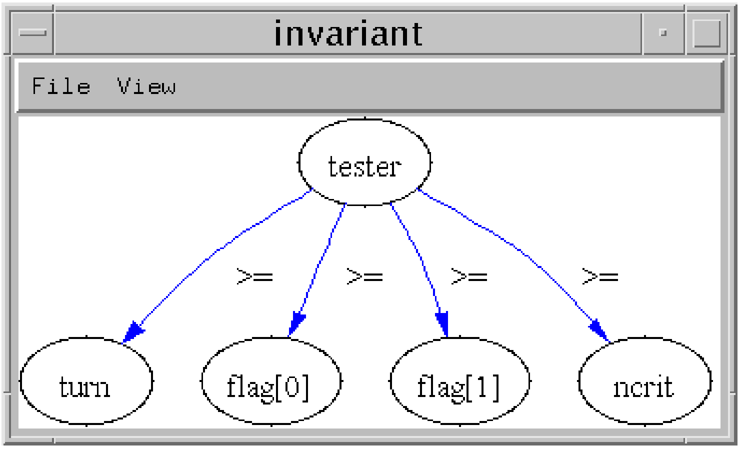

Figure 3: Output of Peterson's Algorithm

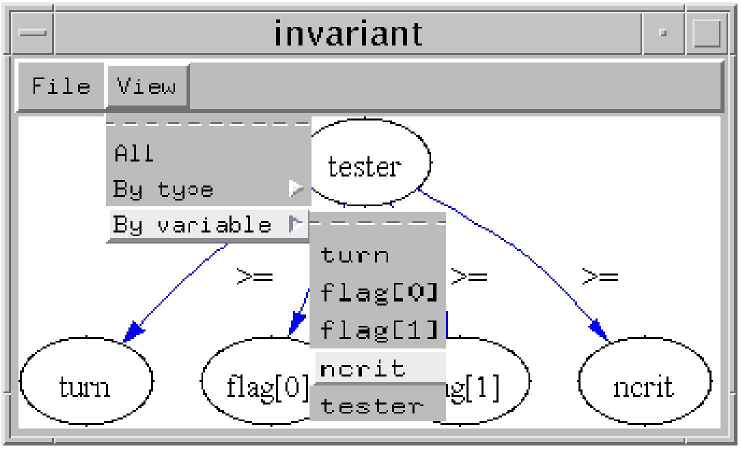

Figure 4: Output of Peterson's Algorithm - View

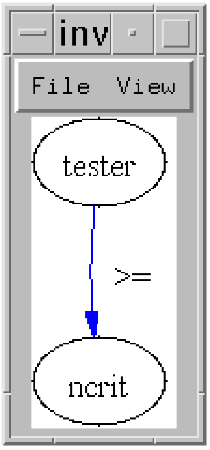

Figure 5: Output of Peterson's Algorithm - View

The invariants may also prove useful when combining model checking and theorem proving. They can be used either directly by the theorem prover or indirectly. In the latter case, they can help the user to conceptually infer invariants of lower-level implementations of the model being verified.

As part of future work, we plan to build invariant generation programs for other form of invariants. For example, invariants that are of the form $a + b\ rel\ c$ may be useful when a and b represent the number of elements on two different channels, and c is a constant. In addition, we would like to explore directions in which the user inputs the desired form of the invariants and/or the variables to be to be related. We plan to work from examples that are currently being used for theorem proving in the Theory of Distributed Systems group at MIT, such as a group communication service [1].

References

[1] R. DePrisco, A. Fekete, N. Lynch, and A. Shvartsman. *A Dynamic View-Oriented Group Communication Service*, in *Proceedings of the 17th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Puerto Vallarta, Mexico, 1998, 227-236.

[2] G.J. Holzmann. *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1991.

[3] N. Bjorner, A. Browne, Z. Manna. *Automatic generation of invariants and intermediate assertions*, Theoretical Computer Science, 173 (1997) 49-87.

[4] S. Bensalem, Y. Lakhnech, and H. Saidi. *Powerful techniques for the Automatic Generation of Invariants*, in *Proc. 8th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Vol. 1102 (Springer, Berlin, 1996) 323-335.