Symbolic GameChecker

March 14, 2012

1 Introduction

Symbolic GameChecker is a prototype tool developed in Java for checking safety properties of open programs. The tool compiles an open program with infinite integers into a symbolic automaton, which represents the game semantics of the program. The main feature of symbolic automata is that the data is not represented explicitly in it, but symbolically. Along with the tool we have also implemented in Java our own library of classes for working with symbolic automata. We could not just reuse some of the existing libraries for finite-state automata, due to the specific nature of symbolic automata we use.

The symbolic automata generated by the tool is checked for safety, i.e. for reachability of a specified move ($abort.run$). We use the breadth-first search algorithm to find the shortest unsafe play (trace) in the model, which contains the unsafe move ($abort.run$). Then an external SMT solver Yices [1] is called to determine consistency of its condition. If the condition is found to be consistent (satisfiable), the unsafe play is feasible, i.e. there is an assignment of concrete values to symbols which makes the condition true. This unsafe trace is reported as a genuine counter-example. Otherwise we search for another unsafe trace in the automaton. If no unsafe trace is discovered or all unsafe traces are found to be inconsistent, then the term is deemed safe.

2 Examples

We provide a few examples as well as detailed reports of how our tool have executed them. Examples are written in the C-like syntax for the sake of its presumed familiarity to most users. With each example (e.g. prog1.ia), we provide some files (e.g. prog1.txt and prog1-1.ys) as generated by Symbolic GameChecker. The file prog1.txt represents the result of executing prog1.ia on Symbolic GameChecker. It contains the description of the symbolic automaton generated for the given program, and unsafe traces found in the automaton. For each unsafe trace, there is a Yices file which describes the condition of that trace. For example, the file prog1-1.ys corresponds to the

[1] http://yices.csl.sri.com

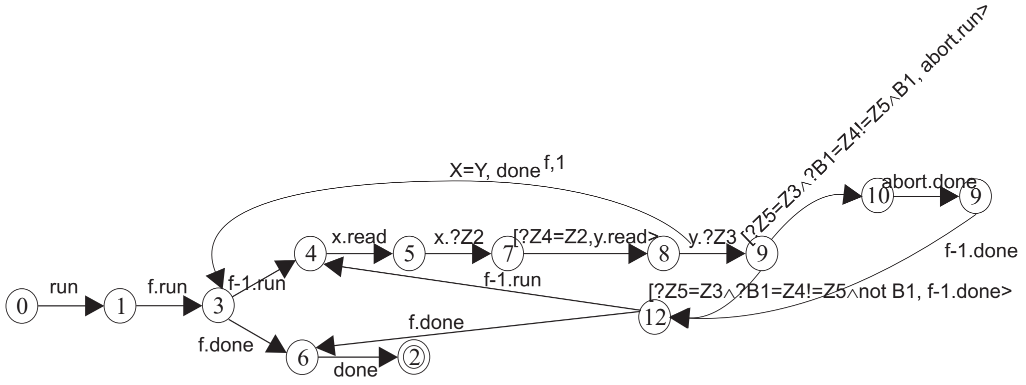

Figure 1: The symbolic model for prog1.ia

shortest (first) unsafe trace of the model for prog1.ia. Those files are executed by the Yices, and the results are used in our tool. Symbolic GameChecker checks unsafe traces until it finds an unsafe trace with consistent (satisfiable) condition.

Let us consider prog1.ia:

$$f : \mathsf{com} \to \mathsf{com}, abort : \mathsf{com}, x : \mathsf{expint}, y : \mathsf{expint} \vdash \\ f\big(\mathsf{if}\ (x \neq y)\ \mathsf{then}\ abort\big) : \mathsf{com}$$

in which f is a non-local procedure, and x, y are non-local expressions. We want to check whether this term is safe from executing *abort*.

The symbolic automaton representing game semantics of this term is generated by Symbolic GameChecker. The automaton is represented by a set of states and a set of transitions leading from one to another state. Each transition is described by a guarded letter $[b, \alpha\rangle$, where b is a boolean condition and α is a symbolic letter. We write α for the guarded letter $[true, \alpha\rangle$. Symbolic GameChecker reports the following automaton for prog1.ia:

$$0\,[run\rangle\,1,\ 1\,[f.run\rangle\,3,\ 3\,[f-1.run\rangle\,4,\ 3\,[f.done\rangle\,6,\ 4\,[x.read\rangle\,5,\ 5\,[x.?Z2\rangle\,7,$$
$$6\,[done\rangle\,2,\ 7\,[?Z4 = Z2, y.read\rangle\,8,\ 8\,[y.?Z3\rangle\,9,$$
$$9\,[?Z5 = Z3 \wedge ?B1 = Z4! = Z5 \wedge B1, abort.run\rangle\,10,$$
$$9\,[?Z5 = Z3 \wedge ?B1 = Z4! = Z5 \wedge notB1, f-1.done\rangle\,12,$$
$$10\,[abort.done\rangle\,11,\ 11\,[f-1.done\rangle\,12,\ 12\,[f-1.run\rangle\,4,\ 12\,[f.done\rangle\,6$$

By using the notation $(2, true)$, it is marked that the state 2 is a final state. Graphically this automaton is shown in Figure 1. The model illustrates only the possible behaviors of this term: the non-local procedure f may call its argument, zero or more times, then the term terminates successfully with *done*. If f calls its argument, arbitrary values for x and y are read from the environment by using symbols $Z2$ and $Z3$. If they are different, then the abort command is executed.

The shortest unsafe trace found in the model is:

$$run\, f.run\, f-1.run\ \ x.read\ \ x.?Z2\ \ [?Z4 = Z4, y.read\rangle\ \ y.?Z3$$
$$[?Z5 = Z3 \wedge ?B1 = Z4! = Z5 \wedge B1, abort.run\rangle\ \ abort.done$$
$$f-1.done\, f.done\ done$$

The consistency of the condition associated with this trace is established by instructing Yices to check the formula found in prog1-1.ys:

$(set - evidence!\, true)$
$(define\, Z2 :: int)$
$(define\, Z4 :: int)$
$(assert\, (=\ Z4\, Z2))$
$(define\, Z3 :: int)$
$(define\, Z5 :: int)$
$(assert\, (=\ Z5\, Z3))$
$(define\, B1 :: bool)$
$(assert\, (=\ B1\, (/=\ Z4\, Z5)))$
$(assert\, B1)$
$(check)$

The result returned by Yices is:

So we can conclude that the term prog1.ia is unsafe, and one concrete counter-example is:

$$run\, f.run\, f-1.run\ \ x.read\ \ x.1\ \ y.read\ \ y.2$$
$$abort.run\ \ abort.done\ \ f-1.done\, f.done\ done$$