Automatic transition trace analysis of parallel programs using VeriSoft

Juergen Dingel

School of Computing
Queen's University, Kingston, Canada
dingel@cs.queensu.ca

Abstract. We show how the state space exploration tool VeriSoft can be used to analyze sequential and parallel C/C++ programs with respect to their transition traces. The analysis determines whether or not a given program can behave as prescribed by some finite transition trace. VeriSoft's exploration is always bounded by a user-specified, finite depth parameter. Therefore, our analysis is guaranteed to terminate, but does not always provide a definite answer. VeriSoft's optimization and visualization techniques make the analysis relatively efficient and effective.

1 Introduction

The use of traditional formal methods typically requires a lot of training and resources. Consequently, there has recently been a lot of interest in making formal methods more light-weight [12]. Compared to traditional formal methods, light-weight formal methods

- give more immediate feedback which allows for a more gradual learning curve and more immediate results,
- offer decidability and efficiency at the expense of expressiveness, and
- are complemented by fully automatic tools which hide the complexities of the method from the novice user.

Examples of light-weight formal methods include systematic state space exploration through model checking [3], object modeling and analysis with Alloy [10, 11], and systematic program testing through VeriSoft [8]. The overall goal of these and other light-weight approaches is to allow even novice users more effective and efficient use of formal techniques and thus pave the way for a more general adoption. The continued industrial success of model checking, for instance, lends a lot credence to this proposal.

Transition traces form an intuitive model of concurrent computation with many pleasant theoretical properties [16, 6, 2]. The use of the transition trace semantics is, however, not light-weight in the above sense. The trace set of a program may be hard to compute or difficult to write down concisely. Moreover, there is no tool support. In this paper, we show how the VeriSoft tool can be used to analyze parallel programs with respect to their transition traces. Our long-term goal is a more light-weight, compositional semantic analysis of concurrent programs.

We proceed by briefly reviewing the transition trace semantics in the next section. Section 3 describes the VeriSoft tool, and Section 4 shows how it can be used to automate aspects of the semantics. Section 5 presents a few examples that illustrate the use of our method. Section 6 concludes and outlines further work.

2 Transition Traces

Any compositional model of concurrent computation needs to take environment behaviour into account. *Transition traces* (sometimes also called *potential* or *partial computations* or *extended sequences*) have proven very useful for the definition of compositional models of fair, shared-variable and message-passing concurrency [16, 6, 2]. The transition trace semantics $\mathcal{T}$ is based on finite and infinite traces of the form

$$(s_0, s'_0)(s_1, s'_1)\ldots(s_i, s'_i)\ldots$$

where each s_i and s'_i is a state, that is, a mapping from program variables to values. A trace represents a possible "interactive" computation of a command in which state changes made by the command (from s_i to s'_i) are interleaved with state changes made by its environment (from s'_i to s_{i+1}). A trace with $s'_i = s_{i+1}$ for all $i \geq 0$ is called *interference-free* or *execution.*

The meaning $\mathcal{T}[\![D]\!]$ of a program D is given by its set of transition traces closed under two conditions called *stuttering* and *mumbling.* These closure conditions correspond to, respectively, the reflexive and transitive closure of a standard "small-step" operational semantics. The semantics models, for instance, sequential composition as trace concatenation, iteration as repeated concatenation, and parallel composition through a fair merge of traces. A hiding operation ensures that local variables are inaccessible outside their scope. In this paper, assignments and expression evaluation will be treated atomically. However, the semantics can easily be extended to handle different levels of granularity (atomicity) [2].

Consider, for instance, the program

```
D_1 ≡ new i = 1 in
        while i < 3 do
          x:=x + 1;
          i:=i + 1
        od
      end
```

The transition trace set $\mathcal{T}[\![D_1]\!]$ of D_1 includes

$$\{(s_0, [s_0|x = s_0(x) + 1])(s_1, [s_1|x = s_1(x) + 1]) \mid s_0, s_1 \in \Sigma_{D_1}\}$$

and

$$\{(s_0, [s_0|x = s_0(x) + 2]) \mid s_0 \in \Sigma_{D_1}\}$$

where Σ_{D_1} denotes the set of all states of D_1. So,

$$(x = 0 \wedge i = 0, x = 1 \wedge i = 0)(x = 5 \wedge i = 3, x = 6 \wedge i = 3)$$
$$(x = 0 \wedge i = 0, x = 2 \wedge i = 0)$$

are two traces of D_1 where $x = v_1 \wedge i = v_2$ stands for the state s in which x has value v_1 and i has value v_2, that is, $s(x) = v_1$ and $s(i) = v_2$. The first trace represents an interactive computation of D_1 in which the environment interferes between the first and second increment of x and changes the values of x and i to 5 and 3, respectively. In the second trace, D_1 executes without interruption. However, the following four traces

$$(x = 0 \wedge i = 0, x = 1 \wedge i = 0)(x = 5 \wedge i = 0, x = 6 \wedge i = 0)(x = 6 \wedge i = 0, x = 7 \wedge i = 0)$$
$$(x = 0 \wedge i = 1, x = 3 \wedge i = 1)$$
$$(x = 0 \wedge i = 1, x = 0 \wedge i = 1)$$
$$(x = 0 \wedge i = 0, x = 2 \wedge i = 2)$$

are not traces of D_1. Note that the last trace is impossible, because it changes the value of the local variable i.

The transition trace semantics is fully abstract with respect to observational equivalence [2]. Moreover, Brookes has also shown how it can be extended to handle procedures by following Reynolds and Oles' possible worlds model [1]. It can be used to validate "laws of parallel programming" like, for instance, the associativity of parallel composition, that is, we have

$$\mathcal{T}[\![[D \parallel D'] \parallel D'']\!] = \mathcal{T}[\![D \parallel [D' \parallel D'']]\!]$$

for all programs D, D', and D''.

Moreover, the semantics supports compositional reasoning about programs and program transformations. Consider, for instance, program D_1 above and programs D_2 and D_3 below.

```
D₂ ≡ new i = 1, t = x in
       while i < 3 do
         t:=t + 1;
         x:=t;
         i:=i + 1
       od
     end
```

```
D₃ ≡ x:=x + 1;
     x:=x + 1
```

While D_1 and D_2 have the same executions, they do not have the same transition traces. A trace distinguishing D_1 and D_2 will be given in Section 5. However, programs D_1 and D_3 have the same transition traces (and thus also the same executions). Consequently, an occurrence of D_1 in some program can, in general, not be replaced by D_2. However, an occurrence of D_1 in some program D can always be replaced by D_3 without changing the behaviour of D.

3 VeriSoft

VeriSoft is a systematic testing tool developed at Bell Labs [8, 9]. It takes as input C/C++ programs in which concurrent threads communicate through a selection of so-called *communication objects* that include semaphores, channels, or shared memory. A thread makes an update to such a communication object visible to the other threads by executing a so-called *visible action.* VeriSoft systematically enumerates all possible sequences of visible actions a program can perform. The maximal length of the paths is limited by a user-defined parameter called *depth.* During the exploration, VeriSoft checks for livelocks (a process has no enabled transition during a sequence of more than a user-specified number of successive states), divergences (a process does not attempt to execute a visible action for more than a user-specified amount of time), deadlocks, and assertion violations. Using the statement *VS_assert(b)*, the user can check whether the boolean expression b is true at a certain location along every execution of the program. Whenever VeriSoft finds the argument of *VS_assert* to be false, it stops and reports an assertion violation error.

VeriSoft works directly on programs written in C/C++. However, only if the program meets a few requirements will VeriSoft be able to guarantee the complete coverage of the state space (up to the specified depth). First, the program must not exhibit nested parallelism, that is, all threads must be created before the initial state is reached. For threads that are created later, VeriSoft's state space exploration may not be exhaustive leading to possibly spurious analysis results. Second, threads need to communicate via the VeriSoft communication objects. If an update to such a communication object is supposed to be visible to the parallel environment, the corresponding visible action needs to be used. Consider, for instance, the following two programs using the shared variable x. In VeriSoft, an update to shared variable x is exported, that is, made visible to the environment, by prefixing it with the visible action *VS_synchro_write("x").*

```
x:=0;                              x:=0;
while true do                      while true do
  VS_synchro_write_file("x");        VS_synchro_write_file("x");
  x:=x + 1;                          x:=x + 1;
  x:=x + 1                           VS_synchro_write_file("x");
od                                   x:=x + 1
                                   od
```

After the initialization, the environment of both programs will find x to be monotonically increasing. The environment of the left program, however, will never be able to find x to be odd, while the environment of the right program is able to access x when it is odd. Note that the visible actions thus determine the granularity of the parallel execution.

Suppose program D meets the two requirements above and D is analyzed with depth d. Then, a successful VeriSoft analysis of D guarantees that the prefix of length d of every execution of D is free of livelocks, divergences, deadlocks, and assertion violations.

VeriSoft keeps the state explosion problem in check by safely pruning the state space. More precisely, it looks for actions that are independent. Consider, for instance, two threads that update a disjoint set of shared variables.

$$x := 0 \mid\mid y := 1$$

Although the program has two different executions, only one of them needs to be considered, because the two assignments are independent. The order in which they are being performed does not change the behaviour of the rest of the program. This optimization is typically called *partial order reduction* and is described in, for instance [7].

VeriSoft facilitates the implementation of test harnesses by supporting non-determinism. The statement *VS_toss(n)* non-deterministically returns an integer between 0 and n inclusive.

VeriSoft allows an entire subtree of the state space to be pruned using *VS_abort(b)* where b is a boolean expression. The execution of *VS_abort(b)* causes b to be evaluated. If b is false, none of the successors of the current state is explored. The use of this statement is similar to the use of *VS_assert(b)* assertion, except that no error is reported. It can be used to prune parts of the state space that the user considers uninteresting.

Finally, a graphical user interface allows paths to be displayed and states to be examined. The tree of all possible paths of the program is shown and the selection of a node in the tree runs the program up to that point and the current values of the variables can be examined. In case of an assertion violation, the path leading to the violating state is displayed. Moreover, the user can steer program execution by stepping through each of the threads, controlling the interleaved execution and determining the result of *VS_toss* operations.

4 Using VeriSoft for transition trace analysis

We want to use VeriSoft to determine whether or not some program D (in some suitably chosen environment) can behave as prescribed by some finite transition trace

$$t \;\equiv\; (s_0, s'_0)(s_1, s'_1)\ldots(s_n, s'_n).$$

More precisely, we want to see whether t is the prefix of a transition trace of D. We assume that the states in t mention exactly the shared variables in D and that D is written in C/C++, uses the VeriSoft communication objects to communicate with its environment and contains neither nested parallelism nor *VS_assert* statements. To check whether or not t is the prefix of a trace of D, we use a C/C++ program $S(D, t)$ that declares the shared variables used in D, sets the initial state to s_0, and then forks two processes. The first process executes D while the other serves as the environment. The purpose of the environment is to monitor all state changes from s_i to s'_i by the program and to carry out the changes from s'_i to s_{i+1}. If the state s'_i created by the program is not equal to the corresponding state in the trace, the environment will abort the execution that led to s'_i. Otherwise, the environment changes the current state to s_{i+1} and executes a visible operation to allow for D to interfere and create s'_{i+1}. This process continues until the entire trace has been matched, D terminates, or the depth of the search is reached. If the entire trace has been matched, we use an assertion violation to signal success. The pseudo code for $S(D,t)$ is given in Figure 1.

```
          $S(D,t)$ $\equiv$ declare all shared variables in D;
                     $i := 0$;
                     initialize state to $s_i$;
                     $[D \parallel E(t)]$

where     $E(t)$   $\equiv$ while true do
                       export current state to allow for D to interfere;
                       if current state $\neq s'_i$ then
                         VS_abort(false);  // abort, process has not created desired state
                       else  // process has created desired state
                         $i := i + 1$;
                         if $i > n$ then
                           VS_assert(false);  // whole trace matched, done
                         else
                           change state to $s_i$;
                     od

  and       $t$     $\equiv$ $(s_0, s'_0)(s_1, s'_1)\ldots(s_n, s'_n)$
```

Fig. 1. Pseudo code for transition trace analysis of program D using VeriSoft

The input-output relation of the analysis of $S(D,t)$ with VeriSoft is as follows. Suppose program D does not contain nested parallelism or *VS_assert* statements and that the level of granularity of D coincides with the level of granularity of $\mathcal{T}$. Let t be a finite trace over the shared variables in D. Finally, suppose VeriSoft is run on $S(D,t)$ with depth d. Three outcomes are possible:

1. VeriSoft reports an assertion violation. In this case, D can behave as prescribed by t, that is, t is the prefix of at least one trace in $\mathcal{T}[\![D]\!]$.
2. VeriSoft completes the exploration of all paths of $S(D,t)$ up to depth d without reporting an assertion violation and D has terminated in the last state of every path. In this case, t is not the prefix of any trace in $\mathcal{T}[\![D]\!]$.
3. VeriSoft completes the exploration of all paths of $S(D,t)$ up to depth d without reporting an assertion violation and D has not terminated in the last state of at least one path. In this case, t may or may not be the prefix of a trace in $\mathcal{T}[\![D]\!]$.

Cases 1 and 2 provide definite answers. Case 3, however, does not. In that case, an increased value of d may allow D to be run to completion and the analysis may terminate in Case 1 or Case 2. However, if D has non-terminating executions, no value of d may be large enough. In that case, the user can at least inspect the state space generated by VeriSoft and thus attempt to get a sense whether or not t is possible for D.

5 Examples

We sketch a few of the analyses we have performed using VeriSoft.

5.1 Interference

Consider the traces

$$\begin{aligned} t_1 &\equiv (x=0, x=1)(x=0, x=1) \\ t_2 &\equiv (x=0, x=2) \end{aligned}$$

Programs D_1 and D_3 have t_1 whereas D_2 does not. All three programs have t_2. Trace t_2 demonstrates that a program state change (s_i, s'_i) can have been brought about by a sequence of multiple visible actions.

5.2 Non-atomic assignment

Although we consider assignments to be atomic in this paper, the effects of non-atomic assignments can be studied through the use of auxiliary local variables. Program D_4 is like D_1 except that the the assignment $x := x+1$ in D_1 is replaced by two assignments $t := x+1\,;\,x := t$ where t is a local variable.

```
D_4 ≡ new $i = 1, t = 0$ in
        while $i < 3$ do
          $t := x + 1$;
          $x := t$;
          $i := i + 1$
        od
      end
```

Note that the environment of D_4 can change the value of x between the completion of the evaluation of $x+1$ in $t := x+1$ and the actual update of x in $x := t$. A trace that illustrates the difference between D_1 and D_4 is

$$t_3 \equiv (x=0, x=0)(x=3, x=1)(x=8, x=4)$$

which is a trace of D_4, but not of D_1.

5.3 Executions

Since executions are special cases of transition traces, our approach can also be used to check if a program has some execution t, that is, whether or not it is capable of running through the states in t when executed without environment interference. Consider, for example, program

$$D_5 \equiv [D_4 \| D_4]$$

We have used our analysis to ascertain that D_5 has execution

$$t_4 \equiv (x=0, x=0)(x=0, x=1)(x=1, x=2)(x=2, x=1)(x=1, x=2)$$

that is, although x is 0 initially and both threads increment x twice, x may be equal to 2 at termination and the value of x may actually decrease intermittently.

5.4 N-process tie-breaker algorithm

For a larger case study, we implemented the n-process tie-breaker algorithm [17] in C/C++ using shared variables. The code is given in Figure 2. The entry

```
TIE(n)  ≡  new in[0..n − 1] = 0, last[1..n − 1] = −1 in [D_0 || D_1 || ... || D_{n−1}]

where for 0 ≤ i ≤ n − 1

    D_i  ≡  while true do
              for l := 1 to n − 1 do                          }
                in[i] := l;                                   }
                last[l] := i;                                 }  entry
                for j := 1 to n st j ≠ i do                   }  protocol
                  while in[j] ≥ l ∧ last[l] = i do skip       }
                od                                            }
              od;                                             }
            cr_i ;   // critical region, always terminating
            in[i] := 0 ;                                         exit protocol
            nc_i   // non-critical region
            od

and where the values of in or last are not changed in cr_i or nc_i        .
```

Fig. 2. Pseudo code for the n-process tie-breaker algorithm

protocol in each process consists of a loop that iterates through $n-1$ levels. A process will only be allowed to enter the critical region, if it has completed all $n-1$ levels. If process D_i is on level $1 \leq l \leq n-2$, that is, $in[i] = l$, it is allowed to advance to level $l+1$, if

– either it is the highest of all processes, that is, $in[j] < in[i]$ for all $0 \leq j \leq n-1$ with $j \neq i$,
– or, it is not the last process to have entered level l, that is, $last[l] \neq i$.

The synchronization conditions in the entry protocol are strong enough to guarantee mutual exclusion and weak enough to ensure deadlock freedom. Eventual entry is guaranteed because the condition that a process i is waiting on will always eventually become true and then remain true until i "moves on" to the next level.

We have used VeriSoft to analyze the transition traces of $TIE(3)$, the 3-process tie-breaker algorithm. Consider the traces t_5 and t_6 in Figure 3. Our

t_5	$in[0]$	$in[1]$	$in[2]$	$last[1]$	$last[2]$
s_0	0	0	0	-1	-1
s'_0	1	0	0	0	-1
s_1	1	0	0	0	-1
s'_1	2	0	0	0	0
s_2	2	0	0	0	0
s'_2	0	0	0	0	0
s_3	0	0	0	0	0
s'_3	1	0	0	0	0
s_4	1	0	0	0	0
s'_4	2	0	0	0	0
s_5	2	0	0	0	0
s'_5	0	0	0	0	0

t_6	$in[0]$	$in[1]$	$in[2]$	$last[1]$	$last[2]$
s_0	0	0	0	-1	-1
s'_0	1	0	0	0	-1
s_1	1	1	0	1	-1
s'_1	1	1	0	1	-1
s_2	1	1	1	2	-1
s'_2	1	1	1	2	-1
s_3	1	2	1	2	1
s'_3	2	2	1	2	0
s_4	2	2	1	2	0
s'_4	0	2	1	2	0

Fig. 3. Traces t_5 and t_6

analysis showed that process D_0 can do t_5, that is, the process can enter its critical region repeatedly, if neither D_1 nor D_2 ever begin their entry protocol. The trace was found at depth 38 after about two minutes and 12 seconds on a SUN Sparc Ultra-250. In trace t_6, process D_0 enters level 1, is first joined by D_1 and then by D_2. Then, D_1 proceeds to level 2 and D_0 does, too. The transition (s_4, s'_4) has D_0 enter and leave its critical region, effectively overtaking D_1. This last transition is impossible for D_0, because D_0 cannot enter its critical region in state s_4. This is because in s_4 process D_0 is not the highest process and also the last process to have entered level 2, i.e., we have $last[2] = 0$ in s_4. Using VeriSoft, the exhaustive exploration of the state space to depth 100 took 2 minutes 47 seconds. None of the traces of D_0 examined during that search was found to have t_6 as a prefix.

6 Conclusion, future work and related work

We have shown how the VeriSoft tool can be used to analyze C/C++ programs with respect to their transition traces or, as a special case, their executions.

We have shown how programs can be analyzed at different levels of granularity (atomicity) through appropriate use of auxiliary local variables and VeriSoft's visible actions. The analysis is sound in the sense that if VeriSoft says that D can exhibit t, then t is indeed the prefix of a transition trace of D. The analysis is incomplete in the sense that it does not always produce a definite answer. In that case, a repeated analysis with a larger depth may provide an answer. The complexity of the analysis grows exponentially with the number of parallel processes and the search depth. However, due to VeriSoft's partial order reduction, the complexity may be substantially better in many cases. The analysis always produces a state space which can be animated and explored using VeriSoft. In our examples, threads communicate through shared memory. Since VeriSoft also supports semaphores and channels, our technique can also be applied to programs that use these means of communication.

6.1 Related work

There is a lot of loosely related work on, for instance, tool support for process algebraic models of concurrency [15, 4, 13], run-time verification and monitoring of software [14] and software model checking [5]. However, we are not aware of any tool support for transition traces.

6.2 Future work

An obvious avenue for future work is to extend the approach to allow for the direct comparison of the trace sets of two different programs. In other words, given two programs D_1 and D_2 the analysis would provide information about whether or not the traces of D_1 are included in the traces of D_2.

Another possible application of VeriSoft is the partial automation of compositional proof systems for parallel programs using the assume/guarantee (also called rely/guarantee) paradigm. For instance, Stirling uses assume/guarantee specifications of the form

$$[P, \Gamma] \;\; D \;\; [Q, \Delta]$$

where P and Q are predicates and Γ and Δ are sets of predicates [18]. The above specification expresses that if program D is run from an initial state satisfying P and in an environment that preserves all predicates in Γ then every final state of D will satisfy Q and every transition of D will leave all predicates in Δ unchanged. This kind of specification allows the formulation of a compositional proof system for parallel programs. It would be interesting to see to what extend VeriSoft could be used to automate this kind of assume/guarantee reasoning.

References

1. S.D. Brookes. The essence of parallel Algol. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1996.

2. S.D. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, June 1996.
3. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
4. R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The concurrency factory: A development environment for concurrent systems. In R. Alur and T. Henzinger, editors, *Computer-Aided Verification (CAV '96), volume 1102 of Lecture Notes in Computer Science*, pages pages 398–401, New Brunswick, NJ, July 1996.
5. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, June 2000.
6. F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm of asynchronous communication. In *Second International Conference in Concurrency Theory (CONCUR'91)*, 1991.
7. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem.* Springer-Verlag, January 1996.
8. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages pages 174–186, Paris, January 1997.
9. P. Godefroid. Software model checking in practice: An industrial case study. In *Proceedings of International Conference on Software Engineering (ICSE'02)*, Orlando, May 2002.
10. D. Jackson. Automating first-order relational logic. In *ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE '00)*, November 2000.
11. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE/ESEC '01)*, September 2001.
12. D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, April 1996.
13. K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Springer International Journal of Software Tools for Technology Transfer 1(1+2)*, 1997.
14. I. Lee, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Monte Carlo Resort, Las Vegas, Nevada, USA, July 1999.
15. Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.
16. D. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications*, LNCS 86, pages 504–526. Springer Verlag, 1979.
17. G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, June 1981.
18. C. Stirling. A generalization of Owicki-Gries' Hoare logic for a concurrent while language. *Theoretical Computer Science*, 89:347–359, 1988.