A Framework for Exogenous Stateless Model Checking

Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte

Department of Informatics, University of Oslo, Norway
{larstvei,einarj,rudi}@ifi.uio.no

1 Introduction

Different execution paths for sequential, deterministic programs are typically explored by means of unit tests, following standard industrial practice. In contrast, concurrent and distributed programs exhibit nondeterminism due to, e.g., message delays, message re-ordering and race conditions. To explore the different execution paths of such programs, one needs to solve issues of *controllability*, to guide the program execution along a desired path, and *state-space explosion*, since the number of possible executions can be very large.

Stateless model checking is a technique to explore the execution paths of a concurrent program [5]. While highly concurrent programs have a large number of execution paths, making full path exploration infeasible, many operations in a program are completely independent and can be reordered, resulting in paths that are equivalent. Only exploring one path per equivalence class yields a significant reduction in the number of executions that need to be explored. Partial order reduction [2] is a general technique that relates dependent events, based on a *happens-before* relation, and considers paths with the same *happens-before* as equivalent and thereby as members of the same *Mazurkiewicz trace* [7]. *Dynamic partial order reduction* [4] is an algorithm for stateless model checking guaranteed to execute at least one path per Mazurkiewicz trace. A variation of dynamic partial order reduction has been shown to achieve optimal reduction [1]. However, dynamic partial order reduction uses backtracking, which is challenging to parallelize. In contrast, *Offline stateless model checking*, developed by Huang [6], based on maximal causality reduction for shared memory systems, allows parallel path exploration and only requires forward execution.

We present a general framework for offline stateless model checking with partial order reduction, which allows parallel path exploration. Our work abstracts from a specific programming language by capturing the language semantics and path equivalence in terms of execution traces.

2 Traces and Relations over Events

We consider a programming language where the execution of a program can be abstractly captured as a trace τ, meaning a sequence of events $e_1 \cdots e_n$ from a (possibly infinite) set of events $\mathcal{E}$, and assume an associated runtime environment (or simulator) that can both emit such a trace and deterministically reproduce an execution that follows a given trace. If a trace τ represents only a prefix of a complete execution, we assume that after following τ the runtime continues non-deterministically until termination, producing a new trace.

Given some τ emitted from the runtime of a program, we are interested in the different *seed traces* that lead the execution down a path distinctly different from the one represented by τ. These seed traces can be computed from three relations over the set of events $\mathcal{E}$:

$e_i \overset{MHB}{\Longrightarrow} e_j$	if the event e_i *must* happen before e_j in all feasible executions
$e_i \oslash e_j$	if the order of e_i and e_j *may* affect the result of an execution.

$e_i \overset{HB}{\Longrightarrow}_\tau e_j$ if e_i happened before e_j in the trace τ.

For a programming language with a trace-based semantics, $\overset{MHB}{\Longrightarrow}$ can be stated at the level of the language semantics, $\oslash$ at the program level, and $\overset{HB}{\Longrightarrow}_\tau$ at the execution level. Intuitively, $\overset{MHB}{\Longrightarrow}$ ensures that the produced seed traces are possible according to the language semantics and $\oslash$ is used to establish equivalence between traces; e.g., a $\oslash$ relation that relates all events with each other will result in exploring all execution paths, while an empty $\oslash$ will result in only running a single execution. Lastly, the $\overset{HB}{\Longrightarrow}_\tau$ relation encodes the trace τ as a relation.

3 An Algorithm for Path Exploration

We present an algorithm for state exploration, where the execution of a program and the search for new execution paths are separated. Our algorithm is formulated as two procedures (see Algorithm 1 and 2); one that interacts with a runtime and keeps track of the state of the search, and one that generates new seed traces from a given explored trace.

Algorithm 1 Trace Exploration

```
1: procedure Explore(Exec, P)
2:     E ← ∅
3:     Q ← {ϵ}
4:     while τ_s ∈ Q \ E do
5:         τ ← Exec(P, τ_s)
6:         E ← E ∪ PrefixesOf(τ)
7:         Q ← Q ∪ GenerateSeeds(τ)
8:     end while
9:     return E
10: end procedure
```

Algorithm 2 Seed Generation

```
1: procedure GenerateSeeds(τ)
2:     R ← ∅
3:     for ⟨e_i, e_j⟩ ∈ ⊘_τ do
4:         ⊘'_τ ← (⊘_τ \ {⟨e_i, e_j⟩}) ∪ {⟨e_j, e_i⟩}
5:         τ' ← Satisfy(φ_{⟹MHB} ∧ φ_{⊘'_τ})
6:         if τ' = τ_s · e_j · τ_r then
7:             R ← R ∪ { τ_s · e_j }
8:         end if
9:     end for
10:    return R
11: end procedure
```

The search starts by exploring an arbitrary run of a program P, which corresponds executing the empty seed trace $Exec(P, \epsilon)$. From this trace, it generates new seed traces, that can be executed in any order. By keeping track of what executions have been explored (and their prefixes), we guarantee not to explore any execution twice. The search terminates when no seed trace is unexplored.

When generating new seed traces, we only consider pairs in $\oslash$. We define $\oslash_\tau = \overset{HB}{\Longrightarrow}_\tau \cap \oslash$, i.e. the interfering pairs in some particular trace τ, and call such pairs a *conflict*. For all conflicts, the algorithm attempts to reverse the order of these two events, while maintaining the order of all other conflicts. If we can find a trace, which respects the $\overset{MHB}{\Longrightarrow}$ relation and the reversed conflict, then we add it to the set of seed traces.

Concretely, we obtain the seed traces by encoding the relations as a SMT (Satisfiability modulo theories) problem, and solving each instance with Z3 [3]. Formulas are encoded as conjunctions of constraints $V(e_i) < V(e_j)$, where V is a mapping from events to integer variables. We only keep the prefixes of the traces generated by Z3 up to the reversed conflict, because we cannot generally make assumptions about what executions are possible after this point in the trace.

Several details are omitted in the presented algorithm, like measures to reduce regeneration of the same seed traces, minimizing the length of seed traces and comparing prefixes by their conflicts. An implementation of the full algorithm is available at https://github.com/larstvei/trace-exploration.

4 Example

Consider a simple shared memory language, where a process may atomically write a value to a variable, or read the current value of a variable. In this language, there is a direct correspondence between the statements of a program and the events of a trace. The $\overset{MHB}{\Longrightarrow}$ relation only need to ensure the thread-local ordering of events. If two events e_i, e_j operate on the same variable, and one of the operations is a write, then we say $e_i \oslash e_j$. The $\overset{HB}{\Longrightarrow}$ relation will relate an event e_i to e_j if e_i happened before e_j in the sequential trace.

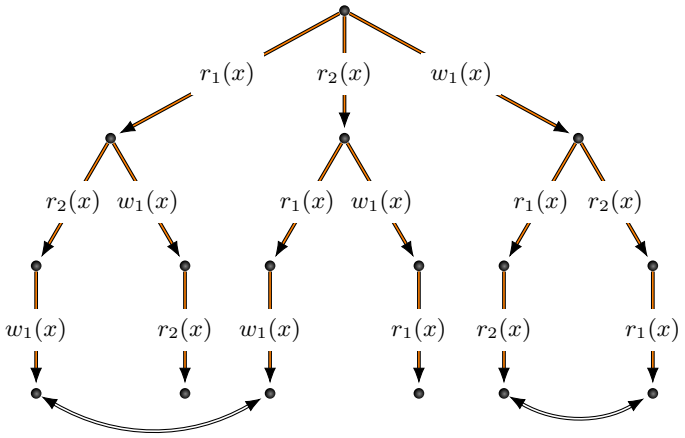

Figure 1: All executions of a simple read/write program.

Consider a simple program with three processes, where each process does one read or write operation, $r_1(x)$, $r_2(x)$ and $w_1(x)$ respectively. We show the possible executions of this program in Fig. 1, where the double-headed arrows indicate an equivalence with $\oslash$ relation as described above. The prototype implementation the yields four non-equivalent traces using this $\oslash$ relation; if we provide one where all events are in conflict, it outputs all the six execution paths.

5 Conclusion and Future Work

The presented framework for exogenous stateless model checking abstracts from any specific programming language and only considers relations over events, as described in Section 2. We showed how to instantiate the framework for a very simple read/write language. In future work, we plan to study the limitations of the framework in order to answer questions about optimality. For example, it is interesting to investigate a possible generalization to different notions of causality (e.g. can the framework be used for both partial order and maximal causality reduction [6]). Furthermore, we want to leverage the framework for a rich actor-based language and evaluate a fully parallelized implementation on complex programs.

References

[1] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proc. of 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '14*, pages 373–384. ACM, 2014.

[2] E. M. Clarke, O. Grumberg, M. Minea, and D. A. Peled. State space reduction using partial order techniques. *Int. J. Softw. Tools Techn. Transfer*, 2(3):279–287, 1999.

[3] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proc. of 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, volume 4963 of*Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '05*, pages 110–121. ACM, 2005.

[5] P. Godefroid. Model checking for programming languages using Verisoft. In *Conf. Record of 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '97*, pages 174–186. ACM, 1997.

[6] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proc. of 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '15*, pages 165–174. ACM, 2015.

[7] A. W. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Proc. of Advanced Course on Petri Nets 1986, Part II*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1987.