TravMC2: Higher-Order Model Checking for Alternating Parity Tree Automata

Robin P. Neatherway
robin.neatherway@cs.ox.ac.uk

C.-H. Luke Ong
lo@cs.ox.ac.uk

Department of Computer Science
University of Oxford
Wolfson Building, Parks Road, OX1 3QD
United Kingdom

ABSTRACT

Higher-order model checking is the problem of model checking (possibly) infinite trees generated by higher-order recursion schemes (HORS). HORS are a natural abstract model of functional programs, and HORS model checkers play a similar rôle to checkers of Boolean programs in the imperative setting. Most research effort so far has focused on checking safety properties specified using trivial tree automata i.e. Büchi tree automata all of whose states are final. Building on our previous work, we present a higher-order model checker, TRAVMC2, which supports properties specified using alternating parity tree automata (or equivalently monadic second order logic). Our experimental results offer an encouraging comparison with an existing checker, TRECS-APT.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Algorithms, Verification

Keywords

Model-checking, Higher-order Programs

1. INTRODUCTION

Model checking has been applied with great success to first-order imperative programs, but only recently has started to gain traction in the verification of higher-order functional programs. In this context the problem of model checking *higher-order recursion schemes* (HORS) should be viewed as a smooth generalisation of finite-state and pushdown model checking, with finite-state and pushdown systems (or Boolean programs) being captured by order-0 and order-1 HORS respectively.

A wave of practically motivated results starting with TRECS [2, 3, 6, 11, 14] has attacked the problem of checking properties described by the class of (alternating) *trivial tree automata*. These are Büchi automata (over possibly-infinite trees) all of whose states are accepting; they correspond to *safety* properties, such as reachability, which have finite counterexamples. Abstraction-refinement techniques [8,13] have enabled the application of these model checkers to Turing-complete programs.

To date, the most efficient safety higher-order model checker, PREFACE [14], readily scales to handle several thousand function definitions, but this is not yet the case for *liveness* properties. Currently there is only a single model checker for liveness properties in the literature, which extends the successful approach of TRECS to properties specified using alternating parity tree automata (APT) [5]. In this paper we present a sound and complete tool that solves the HORS/APT model checking problem by implementing a different, traversal-based, algorithm, and offer an experimental evaluation including a comparison with TRECS-APT.

SPIN'14, July 21–23, 2014, San Jose, CA, USA
ACM 978-1-4503-2452-6/14/07
http://dx.doi.org/10.1145/2632362.2632381

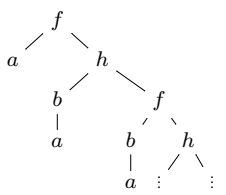
2. IMPLEMENTATION

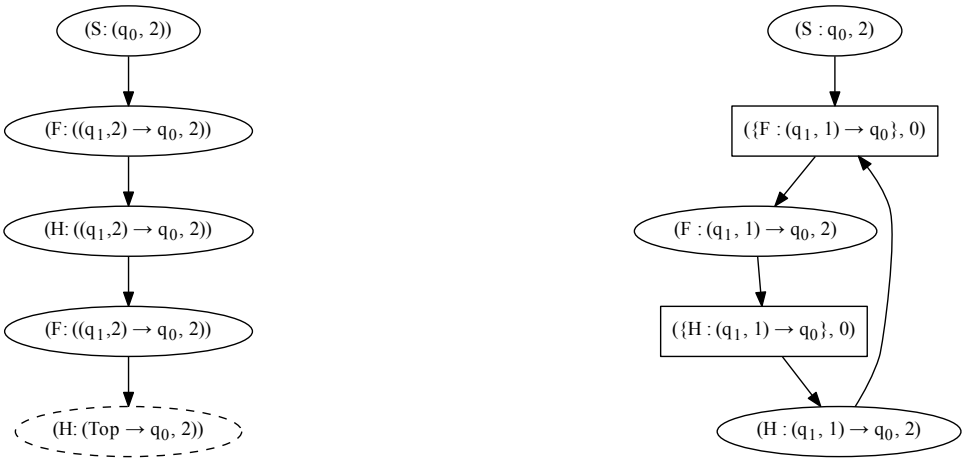
Our tool, TRAVMC2, builds on earlier work [11] for properties specified using deterministic automata with only a single (accepting) priority. In this work we allow full APT which generalise the Büchi acceptance condition: every state is assigned a priority (from a finite set of numbers) and along every infinite path in the input tree, of the states visited infinitely often, the largest priority must be even. Thus, our extension is two-fold: (i) we allow alternation in the transition function of the automaton, and (ii) we allow arbitrary use of priorities. This class of automata, APT, is extremely expressive: it subsumes CTL* and is known to be equi-expressive with monadic second-order logic and modal μ-calculus over trees [4].

We make use of the characterisation of the model checking problem as an intersection type inference problem as given by Kobayashi and Ong [7]. In this setting we take as base types the states of a given property automaton; and a typing judgement $\Gamma \vdash t : q$ is valid if, and only if, Γ is "consistent" (w.r.t. HORS reduction) and t reduces to a possibly-infinite tree accepted from the automaton state q. Therefore, given a HORS with start symbol S and automaton with initial state q_0, if we can find a consistent type environment Γ such that $\Gamma \vdash S : q_0$ is provable then the tree generated by the HORS is guaranteed to be accepted by the automaton.

An example HORS $\mathcal{G}$ and (a prefix of) the tree generated by $\mathcal{G}$ is shown below.

$$S \to F\ a$$
$$F\ x \to f\ x\ (H\ (b\ x))$$
$$H\ x \to h\ x\ (F\ x)$$





(a) Derivation tree **(b)** Witness

Figure 1: A terminating state of TRAVMC2

The reduction of $\mathcal{G}$ starts from S and proceeds by reducing any fully applied function symbol according to the rewrite rules, substituting operands for variables as usual:

$$S \to F\ a \to f\ a\ (H\ (b\ a)) \to f\ a\ (h\ (b\ a)\ (F\ (b\ a))) \to \cdots$$

Each subsequent occurrence of f and h will have left subtree $b^n a$ with n increasing and unbounded; as such this tree is not regular. We might wish to verify that every left branch is finite, which can be achieved using a tree automaton. For simplicity we present a deterministic automaton $\mathcal{A}$. $\mathcal{A}$ has alphabet $\{f, h, a, b\}$, states $\{q_0, q_1\}$ (with initial state q_0), priority function $\{q_0 \mapsto 2, q_1 \mapsto 1\}$ (so that q_0 is accepting and q_1 rejecting) and transition function δ:

$$\begin{array}{ll} (q_0, f) \mapsto q_1\ q_0, & (q_0, h) \mapsto q_1\ q_0, \\ (q_1, b) \mapsto q_1, & (q_1, a) \mapsto \mathtt{t} \end{array}$$

The tree will be accepted by $\mathcal{A}$, as the only infinite path is the spine $(f\ h)^\omega$, which will be labelled by state q_0 (having even priority). The transitions for f and h transition to state q_1 for the left child, but these branches are all finite.

As in previous work [11] our algorithm searches for a witnessing type environment Γ by lazily building a series of typing derivations for the right-hand sides of rules following an outermost reduction strategy inspired by the concept of *traversals* (used in the original game-semantic proof of decidability of the model checking problem [12]). The exploration starts with the requirement to demonstrate that S has type q_0 (which would imply that the tree generated by S is accepted from state q_0 of the automaton). For each type binding we assume for a function symbol we also prove that the right-hand side of the corresponding rewrite rule can be assigned the same type in order to maintain consistency. The tool incorporates a termination check, which searches for duplicate types in the tree of derivations, indicating that the "recursive knot" can be tied. A key difference with the earlier work is that the check for termination must also ensure that the acceptance condition of the automaton is satisfied. In the underlying type system this is achieved by tying the notion of consistency to a parity game over types and environments.

A representation of the algorithm state (a tree of typing derivations) for this simple example after four rounds of expansion can be seen in Figure 1a, where each node is labelled by a type binding and automaton priority, and contains a proof (not shown) that the right-hand side of the corresponding rewrite rule can be assigned the type from the binding. We can see here how the arrow type constructor is used in a standard fashion to represent function types, so that the nodes labelled with F contain typing derivations proving that F is a function that takes a tree accepted from state q_1 after seeing priority 2, and returns a tree accepted from state q_0. In the right-hand side of S this is certainly a useful way for F to behave, as it is applied to a (which is accepted from q_1 according to the transition function of $\mathcal{A}$), and S should generate a tree accepted from state q_0. Although we do not have space to give the type system in full, as an example the typing derivation found in the root node is as follows:

$$\frac{\overline{\{F : ((q_1, 2) \to q_0, 2)\} \vdash F : (q_1, 2) \to q_0} \qquad \overline{\emptyset \vdash a : q_1}}{\{F : ((q_1, 2) \to q_0, 2)\} \vdash F\ a : q_0}$$

The edges are dependencies, so assigning a type to F depends on a type for H and *vice versa*, as might be expected from the rewrite rules. An invariant of the algorithm is that every interior node of the tree contains a valid typing derivation (the final node is dashed to indicate it has not yet been explored); and so, given the dependencies expressed in the edge relation in this state, the algorithm terminates. A witness (see Figure 1b) can be extracted – a parity game where Verifier (owning the ellipse nodes) must play a type environment to justify each binding and is guaranteed to have a winning strategy. Here, the second component of each node is the priority and we can see that the only infinite path has maximum priority 2, making it winning for Verifier.

It is a theorem (the proof of which we elide due to space restrictions) that the input is a YES instance if every path from the root of the derivation tree either:

(i) reaches an explored leaf, or

(ii) encounters two nodes labelled with the same type binding, having an even priority *and* there is no node having a larger odd priority between them. This mirrors the acceptance condition of the APT.

Note that we do not solve a parity game explicitly, although if these conditions are satisfied then necessarily we can extract a witness along with a winning strategy from the current derivation tree.

For instances derived from programs, which are typically much larger, the branching factor of the tree will often range up to five or higher, while the number of nodes in the final algorithm state will be hundreds or thousands. We are guaranteed to find a witness for a YES instance due to the finiteness of the space of intersection types for any given instance, which is n-exponential (i.e. tower of

exponentials of height n) in the order of the HORS. A relationship between computation of the algorithm and running the automaton over the generated tree ensures that the maximal priority labellings of nodes in the derivation tree will eventually be even.

In the "trivial" case, where all automaton states are accepting and therefore counterexamples must be finite, we are also guaranteed to find a witnessing counterexample for a NO instance. However, in this richer setting, we must recover completeness by running our semi-decision procedure for both the property and its complement in parallel.

In order to handle disjunctive choices in the automaton transition function, TRAVMC2 still builds only a single tree as in the deterministic case, but includes path information in the typing derivations and node labels. When checking for termination we consider deterministic projections of the state corresponding to a particular series of choices made by the automaton. If any projection satisfies the termination condition, then we are done.

3. RESULTS

We have benchmarked TRAVMC2 against TRECS-APT using a number of examples, including all those from the TRECS-APT paper [5]. We have modified other examples, abstractions of real programs, from the literature [6, 14] to check liveness properties. Typically this involved strengthening existing properties to ensure that finite behaviours are checked, where before they were not. The tests were carried out on a 2.6GHz Intel Core 2 Quad processor running Windows 7 and the results can be seen in Table 1. TRAVMC2, written in F#, was compiled to a native image using the .NET Ngen.exe facility to avoid the start-up overhead associated with the JIT compiler. For each problem instance we give its size ('S') in terms of the number of symbols in the HORS, the maximum order ('O') of any function symbol, the number of priorities ('P') in the property automaton and the result ('R') i.e. whether the problem was a YES instance. The timings given in seconds; when a tool did not terminate within 60 seconds this is indicated by "–".

We can see that TRAVMC2 is almost always faster than TRECS-APT across the benchmark suite. In particular, on the larger examples (over size 200) in our collection, TRAVMC2 does seem to outperform TRECS-APT consistently. Both tools time-out on examples past a certain size, indicating that more tuning and optimisation is required.

3.1 Optimisations

Without a number of optimisations, the hyper-exponential complexity of the problem is overwhelming, even for relatively small inputs. However, with the following optimisations in place, the results are quite encouraging, handling inputs of up to order 6 and up to 5 priorities.

Subgoal guidance.

When searching for a terminating state between rounds of exploration and enlargement of the derivation tree, along some paths from the root we will find duplicate accepting type bindings as described earlier. In the case where every path has such duplicates we are ready to return YES, otherwise we mark any unexplored nodes on these paths as "dormant" and do not expand them in the next round. This allows us to focus effort in exploring relevant areas of the derivation tree. Note, however, that a given dormant node is not guaranteed to remain dormant in the next round although we have observed that it is often the case that once a node has been marked dormant it remains so until the algorithm terminates.

Table 1: Benchmarking TRAVMC2 and TRECS-APT

Benchmark	S	O	P	R	T-MC2	TRECS
file	74	3	2	Y	0.09	0.21
imperative	79	3	2	Y	0.11	0.30
imperative-awt	82	3	2	Y	0.11	0.30
gcalloc	84	2	3	Y	0.10	0.07
reverse	95	2	2	Y	0.10	0.07
lock1	100	4	1	Y	0.09	0.09
pgm	122	4	2	N	0.27	0.69
merge	135	2	2	Y	0.13	0.52
homrep	142	3	2	Y	0.44	0.36
bsort	146	2	2	Y	0.11	0.22
intercept	151	4	2	Y	0.13	0.23
intercept-awt	155	4	2	Y	1.74	0.28
var-dwt	160	5	2	Y	0.16	1.57
order5-variant-awt	163	5	2	Y	0.17	2.52
loop-dj-2	181	5	5	N	2.12	0.49
fileocamlc-awt	226	4	2	N	0.36	2.02
twofiles	289	3	3	N	0.17	1.10
twofilesexn	296	3	3	Y	0.20	0.95
fileocamlc	311	4	3	Y	0.33	6.59
merge3	538	2	2	Y	1.02	–
map-plus-one	583	5	2	N	1.14	–
dna	699	2	4	Y	4.28	–
map-plus-one-1	854	5	2	N	31.02	–
search-e-church	1303	6	2	N	1.59	–
fold-right	1855	5	1	Y	–	–

Actual parameter revisit avoidance.

As our algorithm explores the problem space in a way guided by the reduction of the HORS, the tool will encounter many situations where a variable is used non-linearly. Up to order 1, if the two occurrences of the variable have the same return type, then computation can be shared as the term bound to the variable will necessarily use any arguments in the same way. In certain situations this can reduce the size of the derivation tree by an exponential factor. Beyond order 1, however, interaction with the context is not sufficiently restricted for the approach to generalise. For a full description, see [11].

Reification caching.

In order to maintain the relationship between intersection types in the derivation tree, we use *open types* built up from type variables. In contrast to the universally quantified type variables used in ML type systems, in our work a type variable of order n is assigned an intersection of open types of order $n-1$ (or automaton states at order 0). Sharing these type variables between typing derivations allows us to preserve an invariant of consistency of the derivation tree, crucial to soundness, even when the control flow of the algorithm becomes very complex at higher orders. However, reifying these types to concrete intersection types in order to perform the termination check can dominate the runtime of the algorithm. By caching the result of reification for each type variable we can avoid the majority of lookups while preserving correctness by invalidating cache entries in the transitive closure of the dependencies for any type variable that we update.

Trivial path conflation.

Given a non-deterministic automaton, the tool must consider every possible consistent sequence of non-deterministic automaton

choices so far explored. However, in the case where the property is specified using a trivial automaton, it is possible to safely confuse paths of exploration corresponding to different non-deterministic automaton choices. This corresponds to initially treating every automaton transition as conjunctive, such that *all* states on the right-hand side of the transition must be accepting, rather than just some satisfying assignment. Observe that for a trivial automaton, every counterexample must be finite. We record those paths that are known not to be accepted by the automaton and, excluding these, consider the rest of the tree conjunctively during the termination check. As every counterexample is finite, the (incorrect) type information they provide is gradually filtered out until eventually, even if there are infinitely many counterexamples, all those remaining are of such length that the tool is guaranteed to find a consistent prefix of the derivation tree that does not include any bad type information. This optimisation is a powerful tool to combat the many paths of exploration introduced by non-deterministic choices.

Benchmark	S	O	R	T_{opt}	T	TRECS-APT
fileocamlc	218	4	Y	0.62	–	–
fileocamlc2	210	4	Y	0.51	–	–
merge2	454	2	Y	1.78	–	4.43
order5-2	141	5	Y	2.70	–	–
rev	109	2	Y	0.16	37.76	0.41
twofiles	143	4	Y	0.12	0.19	3.19

We investigated the effect of this optimisation and the results can be seen in the table above, which contains a number of examples with disjunctive choices added to the property. Timings for this optimisation are in the column "T_{opt}" compared with the standard algorithm in column "T". The results are quite striking and underline the difficulty of efficiently maintaining the non-deterministic choices independently.

3.2 Related Work

In this paper we have focused on an experimental comparison within the HORS model checking paradigm. All practical HORS model checkers except one [2] make use of the intersection type characterization of the model checking problem, including TRECS-APT and TRAVMC2. The underlying algorithms are however rather different. Our algorithm, which builds on TRAVMC, is based on the notion of traversals induced by the fully abstract game semantics of these schemes, but presented as a goal-directed construction of derivations in the intersection type system. The algorithm in TRECS-APT, on the other hand, extracts possible intersection types from a tree representing the concrete reduction of the scheme. A consequence is that in TRECS-APT a parity game must be explicitly solved after every iteration while in our approach a final state simply implies the existence of a winning strategy.

There are notable alternative approaches to the verification of higher-order programs:

(i) Jhala et al. [15] use an SMT solver and predicate abstraction to infer refinement types for programs in OCaml and Haskell. Lazy reduction semantics in Haskell require proving termination of intermediate computations to stay sound, otherwise divergence allows inference of any type, even though the potentially diverging term may never be reduced. This is achieved by encoding a notion of size into the refinement types themselves.

(ii) Binary reachability [9, 10] aims to verify liveness properties by showing that all pairs of states $s_1 \rightarrow^* s_2$ are related by a disjunctively well-founded relation.

A proper comparison with these approaches would require integration of TRAVMC2 with an abstraction technique, and is left for future work.

4. CONCLUSION

We have presented TRAVMC2, a tool that implements a new higher-order model checking algorithm with respect to alternating parity automata. On a range of benchmarks, we have shown that the performance of TRAVMC2 is generally slightly superior to that of TRECS-APT, particularly on the larger examples. Both, however, will require further optimisations to replicate the success of HORS model checkers that target safety properties. We have made the tool and benchmarks available to download [1].

For future work we intend, based on these experiments, to address the highlighted issue of handling non-deterministic choice. The current approach is rather naïve and much work could be shared between the different choices. It may also be productive to consider methods that only distinguish non-deterministic choices where necessary, as in the HORS safety model checker PREFACE [14]. In the long term we aim to integrate TRAVMC2 into a tool with an abstraction phase, to check properties of Haskell or ML programs.

5. ACKNOWLEDGMENTS

We would like to thank Koichi Fujima and Naoki Kobayashi for their help in benchmarking against the TRECS-APT tool, and Steven Ramsay for helpful discussions.

6. REFERENCES

[1] `http://mjolnir.cs.ox.ac.uk/web/horsapt/`.

[2] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore: a collapsible approach to higher-order verification. In *ICFP*, pages 13–24, 2013.

[3] C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In *CSL*, pages 129–148, 2013.

[4] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *FOCS*, pages 368–377, 1991.

[5] K. Fujima, S. Ito, and N. Kobayashi. Practical alternating parity tree automata model checking of higher-order recursion schemes. In *APLAS*, pages 17–32, 2013.

[6] N. Kobayashi. Model-checking higher-order functions. In *PPDP*, pages 25–36, 2009.

[7] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188, 2009.

[8] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In *PLDI*, 2011.

[9] T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Termination verification for higher-order functional programs. In *ESOP*, 2014.

[10] R. Ledesma-Garza and A. Rybalchenko. Binary reachability analysis of higher order functional programs. In *SAS*, 2012.

[11] R. P. Neatherway, S. J. Ramsay, and C.-H. L. Ong. A traversal-based algorithm for higher-order model checking. In *ICFP*, pages 353–364, 2012.

[12] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90, 2006.

[13] C.-H. L. Ong and S. J. Ramsay. Verifying functional programs with pattern matching algebraic data types. In *POPL*, pages 587–598, 2011.

[14] S. J. Ramsay, R. P. Neatherway, and C.-H. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*, pages 61–72, 2014.

[15] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, pages 209–228, 2013.