DEQ: Equivalence Checker for Deterministic Register Automata

A. S. Murawski[1], S. J. Ramsay[2], and N. Tzevelekos[3]

[1] University of Oxford, UK
[2] University of Bristol, UK
[3] Queen Mary University of London, UK

Abstract. Register automata are one of the most studied automata models over infinite alphabets with applications in learning, systems modelling and program verification. We present an equivalence checker for deterministic register automata, called DEQ, based on a recent polynomial-time algorithm that employs group-theoretic techniques to achieve succinct representations of the search space. We compare the performance of our tool to other available implementations, notably in the learning library RALib and nominal frameworks LOIS and NLambda.

Introduction Register automata [9,17] are one of the simplest models of computation over infinite alphabets. They operate on an infinite domain of data by storing data values in a finite number of registers, where the values are available for future comparisons or updates. The automata can recognise when a data value does not appear in any of the registers or has not been seen so far at all [21].

Recent years have seen a surge of interest in models over infinite alphabets due to their ability to account for computational scenarios with unbounded data. For instance, XML query languages [19] need to compare attribute values that originate from infinite domains. In program verification there is need for abstractions of computations over infinite datatypes [9,7] and unbounded resources, such as Java objects [14] or ML references [16]. More broadly, they have been advocated as a convenient formalism for systems modelling, which fuelled interest in extending learning algorithms to the setting [18,3,1,5,13].

This paper presents DEQ, a tool for verifying language equivalence of deterministic register automata (the nondeterministic case is undecidable [17]). As many of the above-mentioned applications rely on equivalence checking, several implementations are available online for comparison. We compare the performance of our tool to the equivalence routine from RALib [4] (a library for active learning of register automata), and two others, programmed in the LOIS [11] and NLambda [10] frameworks. The equivalence-testing routine coded in NLambda has recently been used as part of an automata learning framework [13].

Our experiments show that DEQ compares favourably to its competitors. At the theoretical level, this is thanks to being based on the first polynomial-time algorithm [15] for the problem. The algorithm improves upon the "naive" algorithm that would expand a register automaton to a finite-state automaton over

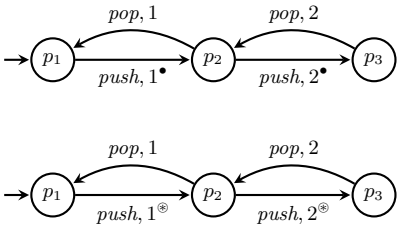

Fig. 1. Two size-2 "fresh" stacks.

```
1 R = R_init; Δ = {u_0};
2 while (Δ is not empty) do
3   u = Δ.get();
4   if u ∉ Gen(R)
5     if u fails 1-step test return NO;
6     Δ.add(succ-set(u));
7     R = R updated with u;
8 return YES
```

Fig. 2. Algorithm in outline.

a sufficiently large finite alphabet, often incurring an exponential blow-up. Ours is also the only tool that can handle global freshness, i.e. the recognition/generation of values that have not been seen thus far during the course of computation. This is an important feature in the context of programming languages that makes it possible to model object creation faithfully.

Register automata Let $\mathcal{D}$ be an infinite set (alphabet). Its elements will be called *data values*. Register automata are a simple model for modelling languages and behaviours over such an alphabet. They operate over finitely many states and, in addition, are equipped with a finite number of *registers*, where they can store elements of $\mathcal{D}$. Each automaton transition can refer to the registers by requiring e.g. that the data value from a specific register be part of the transition's label or, alternatively, that the label feature a *fresh* data value: either not *currently* in the registers, or globally fresh (never seen before), which could then be stored in one of the registers. Formally, transition labels are pairs (t, d), where t is a tag drawn from a finite alphabet and $d \in \mathcal{D}$. We write $q \xrightarrow{t,i} q'$ to specify transitions labelled with (t, d) where t is a tag and d is the data value currently stored in register i. Similarly, $q \xrightarrow{t,i^\bullet} q'$ describes transitions labelled with (t, d), where $d \in \mathcal{D}$ is currently *not* stored in any registers. Once the transition fires, d will be stored in register i. In contrast, $q \xrightarrow{t,i^{\circledast}} q'$ captures transitions with labels (t, d), where d ranges over all elements of $\mathcal{D}$ that have not yet been encountered by the automaton (d is "globally fresh").

Consider the automaton at the top of Figure 1, which simulates a bounded "fresh" stack of size 2. By the latter we mean that the simulated stack can store up to two 2 distinct data values. The automaton starts from state p_1, with all its registers empty (erased). It can make a transition labelled with $(push, d_1)$, for any data value d_1, store it in register 1, and go to state p_2. From there, it can either pop the data value already stored in register 1 and go back to p_1 (also erasing the register), or push another data value by making a transition $(push, d_2)$, for any data value $d_2 \neq d_1$, and go to state p_2. From there, it can pop the data value already stored in register 2 (and erase that register) and go to p_1, and so on.

The other automaton in Figure 1 (bottom) also simulates a 2-bounded fresh stack, but it does so using globally fresh transitions. That is, each $(push, d)$

transition made by the automaton is going to have a data value d that is different from all data values used before. We can thus see that bisimilarity of the two automata will fail after one pop: the upper automaton will erase a data value from its registers and, consequently, will be able to push the same data value again later. The automaton below, though, will always be pushing globally fresh data values. In other words, the following trace $(push, d_1)\,(pop, d_1)\,(push, d_1)$ is permitted by the upper automaton, but not by the lower one.

Implementation We have developed a command-line tool for deciding language equivalence of this class of automata, implemented in Haskell[4]. The two input automata (DRA) are specified using an XML file format (parsed using the *xml-conduit* library [20]). Strictly speaking, the algorithm presented in [15] decides whether two states within the same automaton are bisimilar. Hence, our implementation first transforms the input (an instance of the language equivalence problem for two DRA) into an instance of the bisimilarity problem, by constructing the disjoint union of the two automata.

Our algorithm exploits the observation that (in the deterministic setting) language equivalence and bisimilarity are related, and it attempts to build a bisimulation relation incrementally. To avoid exponential blow-ups, we rely on symbolic representations based on (partial) permutations, which capture matches between register content in various configurations. The outline of the algorithm is presented in Figure 2.

The algorithm is similar in flavour to the classic Hopcroft-Karp algorithm for DFA [8], which maintains *sets of pairs of states*. In contrast, we work with four-tuples (q_1, σ, q_2, h), where q_1, q_2 are states, σ is a partial permutation and h is a parameter related to the number of registers. DEQ represents partial permutations using an implementation of immutable integer maps that is based on big endian patricia trees [12]. Most operations complete in amortized time $O(\mathsf{min}(n, W))$, where W is the number of bits of an integer. This is important because manipulating partial permutations through insertion and deletion is at the core of the innermost loop (line 7).

Starting from a four-tuple u_0, which represents the input equivalence problem, our implementation uses a queue Δ (initialised to $\{u_0\}$). This is used to store the four-tuples that still need to be scrutinised to establish the original equivalence. Since the total number of possible four-tuples is exponential in the number of registers (because one component is a partial permutation over the register indexes), the algorithm prescribes a sophisticated compact representation called a generating system. The generating system $\mathcal{R}$ represents the set of four-tuples that have already been analysed (its initial value $\mathcal{R}_{init}$ contains four-tuples with identical states and identity permutations).

Each iteration of the loop considers a four-tuple u taken from the queue (line 3): first we check if it is already generated by the generating system accumulated so far (line 4). Querying the generating system for membership requires deciding if a permutation belongs to a permutation group generated by $\mathcal{R}$. For this purpose,

[4] The tool and its source are available at http://github.com/stersay/deq.

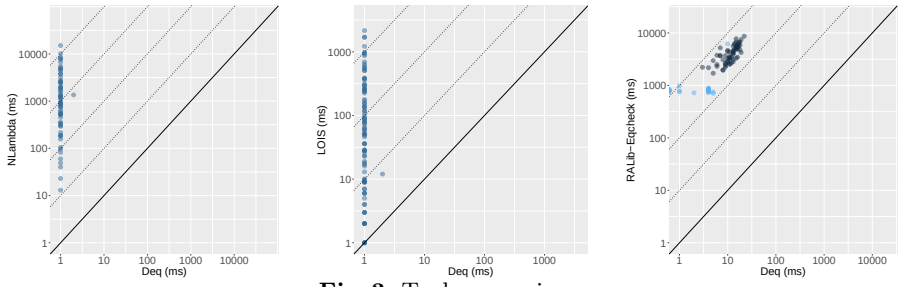

Fig. 3. Tool comparison.

DEQ uses an implementation of the celebrated Schreier-Sims polynomial time group membership algorithm provided by the HaskellForMaths library [2].

If the four-tuple is already generated we move on to the next iteration. Otherwise we check if the configurations represented by u can withstand a one-step attack in the corresponding bisimulation game (line 5). If u fails single-step testing, the algorithm can immediately terminate and return NO. If u passes the tests, the algorithm generates a set $\mathsf{succ\text{-}set}(u)$ consisting of "successor four-tuples" that are added to Δ for future verification (line 6). In this case, the generating system is extended to represent u as well (line 7). For efficiency reasons, DEQ fuses these two parts of the algorithm together. A collection of successors is computed by looping over all outgoing transitions relevant to u, and failing if any cannot be constructed. In what follows we refer to this as the inner loop of the algorithm. The successor four-tuples are then added to the queue and the generating system $\mathcal{R}$ is extended so that it generates u.

Case Studies In theory the worst-case performance of the algorithm is dominated by the complexity of permutation group membership testing, which is $O(n^5)$ for a straightforward implementation of the Schreier-Sims algorithm. In the following series of case studies we examine the performance of our implementation outside of the worst case, in particular where the group structure is quite simple.

All the experiments were carried out on an Ubuntu 16.04 virtual machine, running on a Windows 10 host equipped with an Intel Core i7-8650U CPU at 1.9GHz and 8GB of RAM. Z3 4.4.1 and OpenJDK 1.8.0_191 are used for the purposes of running the tools LOIS and RALib-EqCheck (see below).

Stack data structure (stacks) In this case study, we describe two families of automata simulating finite stacks, indexed by the stack size. Similar families have been considered within the nominal automata learning framework of [13]. In both families of machines, the registers under their natural order are used in order to store the elements of a stack, but one family "pushes" data into the registers from right to left and the other from left to right (so the machines are nevertheless equivalent).

By considering the plot of running time against stack size in Figure 4 (left), we conclude that any overhead due to the group membership algorithm is insignificant when the groups are easy to describe (as growth remains roughly quadratic in n).

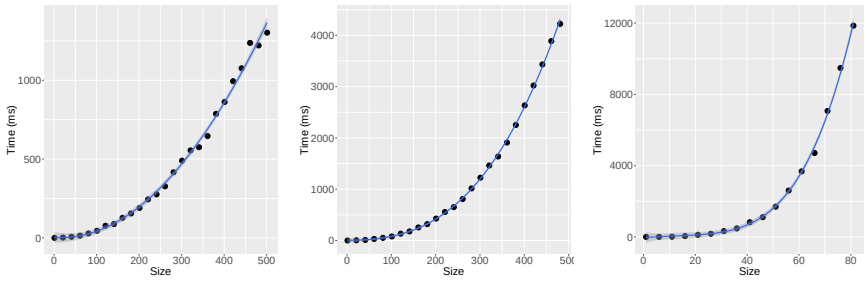

Fig. 4. Tool scaling (stacks, glolo, cpt).

Global simulating local freshness (glolo) In this case study we maintain a relatively simple group structure in the generating system, but introduce globally fresh transitions. Due to the use of global freshness, no other tools will process this example. In this study we consider pairs of machines which first read and store exactly r globally fresh names. Next, one machine reads a locally fresh name, whilst the other reads a globally fresh name, but because all of the history of the computation is stored in registers, local and global freshness coincide, and the configurations are bisimilar. Associated run times are plotted in Figure 4 (middle). It can be seen that the management of the history that is required by supporting global freshness adds a linear factor: the curve displayed is a cubic polynomial, fit using R's *lm* algorithm.

Partial permutations represented compactly (cpt) Here we consider a family of instances in which the collection of partial permutations occurring in constructed bisimulations is large.

In the pairs of automata of this study, all r registers are initially populated over the course of the first r states, but the order in which they are populated may differ. Afterwards, the machines may store a fresh letter in any register or read from any register, and return to the same state. Due to these possibilities, the correspondence between these states in the two machines can become complex. However, since all transitions are available to both automata, they will nevertheless be bisimilar, and hence accept the same language. Running times are plotted in Figure 4 (right); the curve displayed is a fourth-degree polynomial.

Tool comparison We can encode the stacks family of examples using the frameworks of LOIS [11] and NLambda [10], and directly as a set of inputs to RALib-EqCheck[5] [4]. However, these other tools can only handle instances of relatively small size. This is not surprising, since they support classes of automata that are much more general. In contrast, the polynomial time algorithm implemented by DEQ is highly specialised to a specific subclass. Hence, we have restricted our comparison to stacks of size at most 15. The scatter plots in Figure 3 show the running times of the three implementations compared on all possible pairs

[5] We used an unreleased implementation of the equivalence checking algorithm that was kindly communicated to us by F. Howar.

of stack sizes[6]. We can encode the cpt family of examples also as inputs to RALib-EqCheck; the results of comparing the two tools on this family (up to size 5) is shown in light blue in the right-most plot of Figure 3. The results show quite clearly that DEQ can determine (in)equivalence several orders of magnitude faster than the other tools.

Acknowledgement Research funded by EPSRC (EP/J019577/1).

References

1. F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. W. Vaandrager. Learning register automata with fresh value generation. *Proc. of ICTAC*, *LNCS* 9399, 165–183, 2015.
2. D. Amos. http://hackage.haskell.org/package/HaskellForMaths.
3. B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A robust class of data languages and an application to learning. *LMCS* 10(4), 2014.
4. S. Cassel, F. Howar, and B. Jonsson. RALib: A LearnLib extension for inferring EFSMs. *Proc. of DIFTS*, 2015.
5. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.
6. M. L. Furst, J. E. Hopcroft, and E. M. Luks. Polynomial-time algorithms for permutation groups. *Proc. of FOCS*, 36–41, 1980.
7. R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In *Proc. of TACAS*, *LNCS* 7795, 260–276, 2013.
8. J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, 1971.
9. M. Kaminski and N. Francez. Finite-memory automata. *TCS* 134(2):329–363, 1994.
10. B. Klin and M. Szynwelski. SMT solving for functional programming over infinite structures. *Proc. of MSFP*, *EPTCS* 207, 57–75, 2016.
11. E. Kopczyński and S. Toruńczyk. LOIS. *Proc. of POPL*, 586–598, 2017.
12. http://hackage.haskell.org/package/containers.
13. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. *Proc. of POPL*, 613–625, 2017.
14. A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. A contextual equivalence checker for IMJ*. *Proc. of ATVA*, LNCS, 234–240, 2015.
15. A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. Polynomial-time equivalence testing for deterministic fresh-register automata. *Proc. of MFCS*, 72:1–72:14, 2018.
16. A. S. Murawski and N. Tzevelekos. Algorithmic nominal game semantics. *Proc. of ESOP*, *LNCS* 6602, 419–438, 2011.
17. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.
18. H. Sakamoto. *Studies on the Learnability of Formal Languages via Queries*. PhD thesis, Kyushu University, 1998.
19. T. Schwentick. Automata for XML. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.
20. M. Snoyman and A. Breitkreuz. http://hackage.haskell.org/package/xml-conduit.
21. N. Tzevelekos. Fresh-register automata. *Proc. of POPL*, 295–306, 2011.

[6] The encoding used for the comparison with RALib-EqCheck was slightly modified to reflect certain structural constraints imposed by that tool. This alternative encoding is larger and hence runtimes are not comparable with the other two experiments. Note that the timing data for RALib-EqCheck contains JVM start-up time.