Reach on Register Automata via History Independence*

Simon Dierl[1][0000−0001−9730−9335] and Falk Howar[1,2][0000−0002−9524−4459]

[1] TU Dortmund University, Dortmund, Germany
{simon.dierl,falk.howar}@tu-dortmund.de
[2] Fraunhofer ISST, Dortmund, Germany

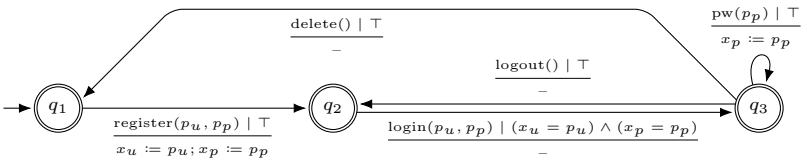
Abstract. Register automata are an expressive model of computation using finite memory. Conformance checking of their properties can be reduced to NonEmptiness tests, however, this problem is PSPACE-complete. Existing approaches usually employ symbolic state exploration. This results in state explosion for most complex register automata. We propose a semantics-preserving transformation of register automata into a representation in which reachability of states is equivalent to reachability of locations, i.e., is in NL. We evaluate the algorithm on random-generated and real-world automata and show that it avoids state explosion and performs better on most instances than a comparable existing approach. This yields a practical approach to conformance checking of register automata.

Keywords: Register Automata · Non-Emptiness · Reachability · History Independence · Constraint Projection · Conformance Checking · Model Checking

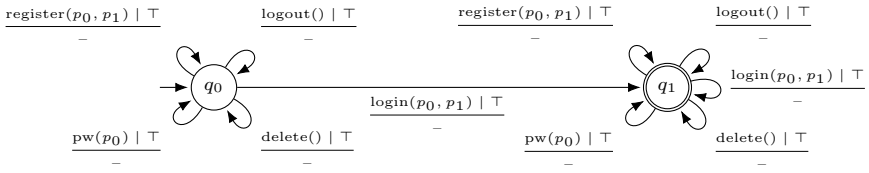
1 Introduction

Register automata (RAs) were introduced by Kaminski and Frances [24] to model languages over infinite alphabets by combining finite-state automata with a finite set of *registers* that can hold data from the inputs. For example, a register automaton can recognize the language of strings beginning and ending with the same letter from an infinite alphabet by storing the initial symbol in a register and comparing every subsequent input symbol to the register. RAs can be used to model many types of real-world systems. As a result, performing model checking on register automata is a relevant problem. Commonly, model checking is performed via conformance checking: a property is modeled as second RA that accepts if the property is satisfied. Then, the product of both automata

* This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-09827-7_2. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms



(a) Model: registration and login



(b) Conformance: at least one successful login

Fig. 1. A RA (a) modeling XMPP [35] account management for a single user [9, Fig. 1] and a RA (b) specifying that at least one login can be performed.

is constructed and tested for NonEmptiness. If it is non-empty, the property is satisfied for at least one input in the original automaton. This is equivalent to running the automaton under test and the specification automaton in parallel and checking if both simultaneously accept.

As an example, Fig. 1(a) shows an RA recognizing the single-user account management fragment of the extensible messaging and presence protocol (XMPP) [35]. In q_1, no account is registered, in q_2, an account exists, but the user is not logged in, while in q_3, the user is. The user's name and password are stored in registers x_u and x_p, respectively. Figure 1(b) specifies that at least one login must be successful by recognizing inputs that contain at least one login() symbol. The language accepted by the product automaton of (a) and (b) is therefore non-empty if the XMPP fragment accepts logins.

However, NonEmptiness (and the more general Reach) are PSPACE-complete for most types of RA (see [16] for a more detailed overview). Two approaches to perform NonEmptiness testing are described in the literature. Sakamoto and Ikeda [36] describe a transformation to equivalent finite-state automata that results in exponential blow-up, while other approaches such as D'Antoni et al.'s configuration LRS [14] for symbolic register automata and Boigelot's symbolic state-space exploration [5] perform symbolic forward search from the initial states, which also creates locations exponential in the number of registers.

To improve upon this, we propose a novel approach that transforms a RA into an equivalent *history independent* RA. In these, it is guaranteed that in every reachable state, a matching input for every subsequent transition exists. Therefore, we can reduce Reach for the automaton's execution states to Reach on the transition graph, which is only an NL-complete problem. We achieve this by back-propagating constraints imposed by a subsequent guard and splitting locations when necessary. Instead of tracking every possible register state, this

approach only considers properties that are enforced by subsequent guards, greatly reducing the number of symbolic states for many automata. We verify this on both real-world and synthetic automata. Our implementation is publicly available on GitHub.[3]

Related Work. Similar to our approach, Garhewal et al. [18] track location constraints in their SL* learning algorithm for register automata. They use forward propagation, and discard constraints that do not discriminate accepted suffixes to avoid exponential blow-up. Iosif and Xu [23] also applied incremental refinenemt in a forwards-propagation algorithm for alternating data automata.

Cassel et al. developed RALib [8], another RA library for the JVM. Since we use slightly different RA semantics, we decided to implement our own RA library. Chen et al. [11] also explore emptiness tests for extended theories; similar approaches are used by [7] and [13].

Constraint solving via SMT solvers on Open pNets was used in [25] to perform dead-transition removal to enable bisimulation checking and in [33] for single-step satisfiability testing by encoding the complete system as a SMT problem.

Moerman and Sammartino studied residuality [27] in nominal automata [6], a property similar to our history independence property.

2 Preliminaries

This section introduces the subset of first-order logic used by register automata and our proposed algorithm, as well as data words and register automata themselves.

2.1 Logical Operations

First, we define the subset of logical formulae relevant to our problem and operations on them.

Definition 1 (Conjunction of Comparisons, Valuation). *Given a set V of* variables*, the logic of* conjunctions (of comparisons) over V $\mathsf{CC}[V]$ *is a subset of first-order logic formulae with variables V. It is defined by*

$$\mathsf{C}[V] ::= (v_1 \circ v_2) \text{ for } \circ \in \{=, \neq\}, v_1, v_2 \in V$$
$$\mathsf{CC}[V] ::= \top \mid \bot \mid \mathsf{C}[V] \ (\wedge \ \mathsf{C}[V])^\star,$$

where $\top$ is universally true and $\bot$ is unsatisfiable. We refer to elements of $\mathsf{C}[V]$ as clauses *and v_1, v_2 as* variables*. The set of* free variables *is empty if $f = \top$ or $f = \bot$ and otherwise is the set of variables present in one or more clauses.*

For a given domain D, a valuation $\phi : V \to D$ satisfies *a conjunction $f \in \mathsf{CC}[V]$ if and only if $f = \top$ or $f = \wedge_{c \in C} c$ and for all $(v_i \circ v_j) \in C$, $\phi(v_i) \circ \phi(v_j)$. We then write $\phi \models f$.*

[3] https://github.com/tudo-aqua/koral

Two conjunctions f, g are equivalent ($f \equiv g$) if the set of satisfying valuations is identical for both.

Our proposed algorithm will make use of two operations on these conjunctions. Given a conjunction of comparisons, a constraint projection generates the strongest statement about a subset of its variables implied by the conjunction. Limiting logical operators to $\wedge$ permits all $\mathsf{CC}[X]$ formulae to be represented as a matrix and the following algorithms to be implemented efficiently.

Definition 2 (Constraint Projection). *Let $f \in \mathsf{CC}[V]$ and let $W \subseteq V$. The* (constraint) projection $\Pi_W(f)$ *is a logical formula such that*

- $f \implies \Pi_W(f)$,
- *all free variables in $\Pi_W(f)$ are $\in W$, and*
- $\forall g$ *with free variables* $\in W$ *and* $f \implies g$, $\Pi_W(f) \implies g$.

For example, $\Pi_{\{x,z\}}(x = y) \wedge (y = z)$ is $(x = z)$, since it is implied by the statements in the original formula. $\Pi_W(\bot) = \bot$ for all W, since it is the strongest possible constraint. In a matrix representation, this operation can be implemented by computing the transitive closure with the Floyd-Warshall algorithm [12] in polynomial time. We next formalize renaming.

Definition 3 (Renaming). *Let $f \in \mathsf{CC}[V]$. The* renaming $f[v'/v]$ *generates a formula in which all instances of v are replaced with v'. For vectors V, V' of equal size n, we write $f[V'/V] = f[v'_1/v_1, \ldots, v'_n/v_n]$.*

2.2 Register Automata

Register automata recognize a combination of a finite and an infinite alphabet. The finite alphabet defines labels that are then combined with values from the infinite alphabet. We now formally define these combinations and mostly follow [9, 10, 16].

Definition 4 (Data Universe, Symbol, Word). *A* data universe *is a tuple $\mathcal{D} = (\Lambda, D, \mathfrak{a})$ with a finite set Λ of* labels*, an infinite set D of* (data) values*, and an* arity *function $\mathfrak{a} : \Lambda \to \mathbb{Z}_{\geq 0}$. For a given label λ, the vector of* formal parameters *is $P^\lambda = (p^\lambda_1, \ldots, p^\lambda_{\mathfrak{a}(\lambda)})$. A* data symbol *is a tuple $(\lambda, \boldsymbol{d})$ with $\lambda \in \Lambda$ and a vector of data values $\boldsymbol{d}$ with $|\boldsymbol{d}| = \mathfrak{a}(\lambda)$. We usually write a symbol as $\lambda(d_1, \ldots, d_{\mathfrak{a}(\lambda)})$. A* data word *is a sequence of data symbols.*

The internal state (q, χ) of a register automaton is defined by its current *location* $q \in Q$ (similar to a finite-state automaton) and the *register valuation* $\chi : X \to D$, i.e., RAs store data values in their registers. Register automata use guard expressions to impose conditions on inputs and their current state. We study guards with equality and inequality comparisons, although in the literature, more expressive guards (e.g., using less-than comparisons) have been discussed. While in the literature, multiple competing formalisms exist, the chosen model subsumes most of them without necessitating expensive transformations (cf. [16]).

Definition 5 (Register Automaton). *A* register automaton (RA) *is a tuple* $\mathcal{A} = (\mathcal{D}, Q, X, S_0, Q^+, \Gamma)$*, defining*

- *a data universe* $\mathcal{D} = (\Lambda, D, \mathfrak{a})$*,*
- *a finite set of* locations Q*,*
- *a finite set of* registers X *that can store data values,*
- initial states $S_0 : Q \times (X \to D)$*,*
- accepting locations $Q^+ \subseteq Q$*, and*
- *a set* Γ *of* transitions $\langle q, q', \lambda, g, u\rangle$*, each defining*
  - *a* source location $q \in Q$*,*
  - *a* target location $q' \in Q$*,*
  - *a label* $\lambda \in \Lambda$*,*
  - *a* guard $g \in \mathsf{CC}[X \cup P^\lambda]$*, and*
  - *an* update $u : X \to (X \cup P^\lambda)$ *that selects new values for the registers visible in the target location, i.e.,* $u(x) \coloneqq v$ *if the value of register or parameter* v *is copied to* x*.*

A transition $\langle q, q', \lambda, g, u\rangle$ is always rendered as

$$\frac{\lambda(p_1^\lambda, \ldots, p_{|\mathfrak{a}(\lambda)|}^\lambda) \mid g}{u},$$

where $p_1^\lambda, \ldots, p_{|\mathfrak{a}(\lambda)|}^\lambda$ are the formal parameters, g is the guard and u is a set of parallel updates $x_i \coloneqq v$ with $v \in X \cup P^\lambda$. If no explicit update to a register x_i is given, the update $x_i \coloneqq x_i$ is implicitly assumed. Next, we will define the execution and acceptance semantics of register automata.

Definition 6 (State Transition). *For a register automaton with a transition* $\gamma = \langle q, q', \lambda, g, u\rangle \in \Gamma$*, a* state transition *is a tuple* $\mathcal{T} = \langle s, s', \gamma, \lambda(d_1, \ldots, d_{\mathfrak{a}(\lambda)})\rangle$*, defining*

- *a* source state $s = (q, \chi)$*,*
- *a* target state $s' = (q', \chi')$*,*
- *an* underlying transition $\gamma = \langle q, q', \lambda, g, u\rangle$*, and*
- *a* data symbol $\lambda(d_1, \ldots, d_{\mathfrak{a}(\lambda)})$ *from* $\mathcal{D}$*,*

such that g *is satisfied under the valuation* $\nu : X \cup P^\lambda \to D$ *defined as*

$$\nu(v) \coloneqq \begin{cases} \chi(v) & \text{if } v \in X \\ d_i & \text{if } v = p_i, \end{cases}$$

and the target valuation χ' *is defined by* $\chi'(x) = \nu(u(x))$*.*

Definition 7 (State Transition Sequence). *Given a register automaton* A*, a* state transition sequence (STS) *is a sequence of state transitions* $\mathcal{T}_1, \ldots, \mathcal{T}_k$ *such that for* $1 \leq i < k$*, the target state of* $\mathcal{T}_i$ *is the source state of* $\mathcal{T}_{i+1}$*.*

The sequence is induced *by the data word formed by the data symbols of each state transition. If the initial state of* $\mathcal{T}_1$ *is* $\in S_0$*, the sequence is* initial*. If the target state of* $\mathcal{T}_k$ *is* (q_k, χ_k) *and* $q_k \in Q^+$*, the sequence is* accepting*.*

Definition 8 (Acceptance Behavior). *A register automaton A* accepts *or* rejects *data words from its data universe. A data word $\lambda_1(\boldsymbol{d}_1)\dots\lambda_k(\boldsymbol{d}_k)$ is* accepted *if it induces an initial and accepting STS. A data word that is not accepted is* rejected. *The language of words accepted by the automaton is $L(A)$.*

A is non-empty if and only if $L(A) \neq \emptyset$. We call the corresponding decision problem NonEmptiness.

Theorem 1. *NonEmptiness is* PSPACE*-complete. [15]*

3 History Independence

We now introduce history independent register automata. Intuitively, a RA is history independent if in each state reachable from an initial state, for every outgoing transition there exists at least one input that enables that transition. As a result, all locations are reachable and transitions can not be "locked out" by register contents. This greatly simplifies analysis of the RA.

Definition 9 (History Independent). *A transition $\gamma = \langle q, q', \lambda, g, u\rangle$ is* history independent *if for each initial state transition sequence ending in (q, χ), there exists a data symbol $\lambda(d_1, \dots, d_{\mathfrak{a}(\lambda)})$ such that there exists a state transition $\langle (q, \chi), (q', \chi'), \gamma, \lambda(d_1, \dots, d_{\mathfrak{a}(\lambda)})$ for a valuation χ'. A register automaton in which every location is history independent is also* history independent.

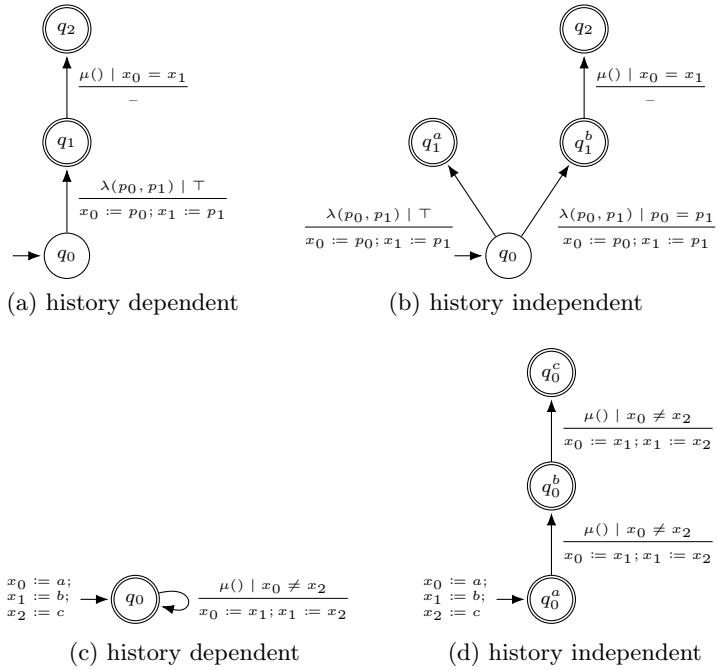
Figure 2 shows examples of history dependent and independent automata. Automaton (a) is not independent, since the state $(q_1, \{x_0 \mapsto a, x_1 \mapsto b\})$ with $a \neq b$ conflicts with the transition guard. In this state, no symbol that continues the state transition sequence along the q_1-q_2-transition may exist. Automaton (b) splits q_1 into a terminal version q_1^a and one, q_1^b, with a stronger guard on the incoming edge. This automaton is independent, since in every state (q_1, χ), $\chi \models x_0 = x_1$ and $\mu()$ continues the sequence. Automaton (c) is dependent since after two iterations, $x_0 = x_2$ must hold and the $\mu()$ transition can no longer be taken. Automaton (d) becomes independent by unrolling the transition. After two steps, not more transitions are available.

History independence simplifies analyses on the automaton, since for every path, a matching input must exist, and reachability in the automaton is equivalent to reachability in its graph structure.

Theorem 2. *A history independent automaton is non-empty if and only if one of its accepting locations can be reached from one of its initial locations in the automaton's transition graph structure.*

Proof. If the automaton is non-empty, an accepting input exists. It induces an STS witnessing reachability in the transition graph.

If an accepting location is reachable in the transition graph, a witnessing path $\gamma_1, \dots, \gamma_k$ exists such that γ_1's source location q_0 has an initial state (q_0, χ_0) Since the automaton is history independent, an input symbol $\lambda(\boldsymbol{d})$ must exist that induces a state transition from (q_0, χ_0) to (q_1, χ_1), where q_1 is γ_1's target. Repeat this argument inductively to arrive at (q_k, χ_k), where $q_k \in Q^+$. □



(a) history dependent (b) history independent

(c) history dependent (d) history independent

Fig. 2. History dependent and independent RAs accepting the same languages. (a) and (b) accept $\lambda(d,d)\mu() + \lambda(d,e)$; (c) and (d) accept $\mu()^{\{0\text{-}2\}}$.

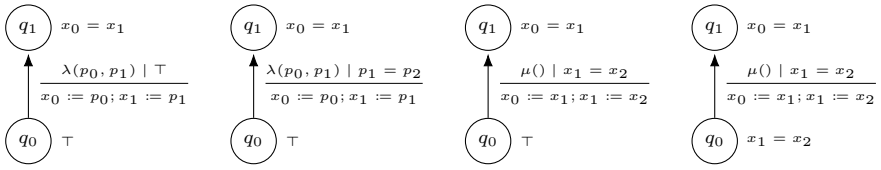
4 Backwards-Propagation Algorithm

This section introduces our algorithm for transforming register automata into equivalent, history-independent RAs. We begin by introducing location constraints and well-constrained transitions, properties that form a statically verifiable, sufficient condition for history independence. We then outline an algorithm to transform one RA transition into a well-constrained transition. Its iterative application yields a second algorithm that transforms an RA into an equivalent, history independent RA. While the transformation causes an exponential increase in size for some inputs, it is designed to limit the blow-up in most cases. We prove both algorithms' correctness and their worst-case complexity.

4.1 Well-Constrained Transitions

Definition 10 (Location Constraint). *Given a register automaton with location set Q,* location constraints *are a function $C_Q : Q \to \mathsf{CC}[X]$. If for each initial state (q, χ) and for each initial STS ending in (q, χ), $\chi \models C_Q(q)$, then C_Q is* valid in q.

The constant $\top$ function $q \mapsto \top$ is a valid location constraint for every RA. We also define a shorthand notation for transforming updates into logical constraints.



(a) ill-constrained (b) well-constrained (c) ill-constrained (d) well-constrained

Fig. 3. Not well-constrained and well-constrained transitions. Constraints are shown next to the locations.

Definition 11 (Update Constraint). *Given an update* $u = x_1 := v_1, \ldots, x_n := v_n$ *with* $v_i \in X \cup P^\lambda$ *for a label* λ*, the* update constraint *of* a *is* $C_U(u) = \wedge_{x:=v \in a}(x' = v)$.

We can now construct a property of location constraints and transition guards that is sufficient for proving history independence. This enables us to statically verify history independence instead of reasoning over all inputs.

Definition 12 (Well-Constrained Transition). *Given location constraints* $C_Q : Q \to \mathsf{CC}[X]$*, a transition* $\langle q, q', \lambda, g, u\rangle$ *is* well-constrained *if*

$$C_Q(q) \implies \Pi_X(g), \text{ and}$$
$$C_Q(q) \wedge g \wedge C_U(u) \implies C_Q(q')[X'/X].$$

Well-constrained transitions are essentially equivalent to Hoare triples [20] of form $\{C_Q(q)\}\gamma\{C_Q(q')\}$. Figure 3 shows two examples of non-well-constrained and well-constrained transitions. (a) is not well-constrained, since the guard does not imply the destination constraint. (b) fixes this by strengthening the guard. Note that this may change the semantics of the transition. In (c), the guard is sufficient, but not implied by the source constraint. This is fixed in (d).

Theorem 3. *Given a register automaton with location set* Q *and location constraints* $C_Q : Q \to \mathsf{CC}[X]$*, a well-constrained transition* $\langle q, q', \lambda, g, a\rangle$ *is history independent if* C_Q *is valid in* q.

Proof. Assume a transition $\gamma = \langle q, q', \lambda, g, a\rangle$, an initial STS ending in (q, χ), and location constraints C_Q valid in q such that γ is well-constrained. $g \not\equiv \bot$, since otherwise $\Pi_X(g) \equiv \bot$ and $C_Q(q) \equiv \bot$, i.e., χ could not exist. Now, we show that a valuation $\nu : (X \cup P^\lambda) \to D$ exists that satisfies g and contains χ (i.e., only the parameters could be set). Consider the formula

$$\ell = \bigwedge_{\substack{x,x' \in X \\ \circ \in \{=,\neq\}}} \{(x \circ x') \mid \chi(x) \circ \chi(x')\},$$

which is satisfied by χ. If ν does not exist, $\ell \wedge g$ must be unsatisfiable. Since $g \not\equiv \bot$, the contradiction must pertain to the registers X, i.e., $g \implies (x_i = x_j)$

and $\ell \implies (x_i \neq x_j)$ or vice versa. Then, however, $\Pi_X(g)$ contains $(x_i = x_j)$ and $C_Q(q) \implies g$, so χ can not satisfy $C_Q(q)$. Therefore, a ν extending χ must exist and can be used to select parameter values for P^λ for a state transition. □

Well-constrained transitions also ensure a propagation of valid location constraints.

Theorem 4. *Given location constraints $C_Q : Q \to \mathsf{CC}[X]$ and a state transition $\langle (q, \chi), (q', \chi'), \gamma, \lambda(\boldsymbol{d}) \rangle$ with γ being a well-constrained transition, then, if χ satisfies $C_Q(q)$, χ' satsifies $C_Q(q')$.*

Proof. By contradiction. Assume an initial STS ending in $\langle (q, \chi), (q', \chi'), \gamma = \langle q, q', \lambda, g, u \rangle, \lambda(\boldsymbol{d}) \rangle$, and location constraints C_Q such that γ is well-constrained, $\chi \models C_Q(q)$ and $\chi' \not\models C_Q(q')$. Then, there must exist x_1, x_2 such that $\chi'(x_1) \neq \chi'(x_2)$, but $C_Q(q') \models x_1 = x_2$ (or vice versa). Now, let $v_1 = a(x_1)$ and $v_2 = a(x_2)$. Since the transition is well-constrained, $g \implies v_1 \neq v_2$, i.e., the transition can not have been taken. □

This informs the central idea of our history-independence transformation: if we can replace all transitions with semantically equivalent well-constrained transitions without altering the automaton's semantics, the resulting automaton is history independent. We will first demonstrate how to make a transition well-constrained using two steps: modifying the guard and updating the source location constraint to make the transition well-constrained and splitting locations to handle incompatible constraints. The first step is straightforward: given a transition $\langle q, q', \lambda, g, u \rangle$ and location constraints C_Q, we can make the transition well-constrained by refining the guard g and location constraint $C_Q(q)$ to

$$\begin{aligned} g' &\coloneqq g \wedge \Pi_{X \cup P^\lambda}(C_U(u) \wedge C_Q(q')) \\ C_Q(q)' &\coloneqq C_Q(q) \wedge \Pi_X(g). \end{aligned}$$

If two transitions $\langle q, q'_1, \lambda_1, g_1, a_1 \rangle$ and $\langle q, q'_2, \lambda_2, g_2, a_2 \rangle$ originate in the same location q, no $\mathsf{C}_Q(q)$ may exist that makes both transitions well-constrained. In this case, we must split the location into multiple variants, each with the same incoming transitions and loops. To preserve possible determinism, each split would create at least four locations:

- a location q_{12} with $C_Q(q_{12}) = \Pi_X(g_1) \wedge \Pi_X(g_2)$, in which both transitions are present and well-constrained,
- two locations $q_{1\bar{2}}$ and $q_{\bar{1}2}$ with $C_Q(q_{1\bar{2}}) = \Pi_X(g_1) \wedge \Pi_X(\neg g_2)$ and vice versa, in which only one transition is present and well-constrained, and
- a location $q_{\bar{1}\bar{2}}$ with $C_Q(q_{\bar{1}\bar{2}}) = \Pi_X(\neg g_1) \wedge \Pi_X(\neg g_2)$, in which no transition is present.

However, the resulting constraints may not be expressible in $\mathsf{CC}[X]$ if they contain $\vee$ operators. Admitting these operators would preclude efficient implementation, so the locations must instead be split into sub-locations for each disjunctive clause. This will frequently result in exponential blow-up. Instead, we sacrifice determinism and create only three locations:

- two locations q_1 and q_2 with $C_Q(q_1) = \Pi_X(g_1)$ and vice versa, in which only one transition is present and well-constrained, and
- a location $q_\top$ with $C_Q(q_\top) = \top$, in which no transition is present.

Note that if a location is split multiple times, $q_\top$ does not need to be recreated. Loop transitions pose an additional challenge, since the location constraint required to make it well-constrained may vary between loops. To preserve semantics, a loop $\langle q, q, \lambda, g, u\rangle$ must be split into two transitions:

- $\langle q_l, q, \lambda, g' = \Pi_{X\cup V}(C_U(u) \wedge C_Q(q)), u\rangle$ with $C_Q(q_l) = \Pi_X(g')$, which is well-constrained, and
- $\langle q_l, q_l, \lambda, g, u\rangle$, which may not.

However, the new loop can again be split iteratively until a stable state is reached. We will prove this intuition in the next section in Lemma 1. We formalize our idea as Algorithm 1. Figure 4 shows two examples of the algorithm's operation. In (a) and (b), q_1' is created to accommodate the μ transition. The guard and the location constraint of q_1' are strengthened so the transition is now well-constrained. Note that the new q_0-q_1' transition is not well-constrained. In (c) and (d), loop handling is shown. The unrolled iteration (the q_1'-q_1 transition) is well-constrained, the copied loop is not.

Note that the loss of determinism is significant, since deterministic RAs are a strict subset of nondeterministic ones. E.g., the former are closed under complement and their Universality is decidable, while nondeterministic RAs are not closed under complement and Universality is undecidable. This does not impact our Reach analysis, however, which will benefit from the transformation. Next, we prove that the algorithm does not alter an RA's semantics, conditional on the validity of the location constraints.

Theorem 5. *Let A be an RA with location constraints C_Q and A^* the RA with location constraints C_Q^* after applying Algorithm 1 to a single transition $\gamma = \langle q, q, \lambda, g, u\rangle$. A data word induces an accepting initial STS in A such that in each state $(\bar{q}, \bar{\chi})$, $\bar{\chi} \models C_Q(\bar{q})$ if and only if it induces an accepting initial STS in A^* such that $\bar{\chi} \models C_Q^*(\bar{q})$.*

Proof. State transition sequences may include three types of modified transition. For each type, we show that an original transition can be exchanged with the modified transition and vice versa.

Outgoing transitions For STS in A^*, they can be replaced by the original transition with a weaker guard. For STS in A, assume a state transition $\langle(q, \chi), (q', \chi'), \gamma, \lambda(\boldsymbol{d})\rangle$. Since $\chi' \models C_Q(q')$ and the original guard is satisfied, the stronger guard created in Line 4 must also be satisfied, so the transition can be replaced with the modified variant.

Loop transitions For STS in A^*, they can again be replaced with the original. For STS in A, the loop may have been split. Then, all but the penultimate loop iteration can be replaced with the loop copy made by Line 9; the last iteration then uses the transition to the original location. If not, but the

```
1  Function MakeWellConstrained(A, C_Q, γ = ⟨q, q', λ, g, u⟩) is
2      remove γ from A;
3      SplitLocation(A, C_Q, q, ⊤);
4      g* := Π_{X∪V}(a ∧ C_Q(q'));
5      c := Π_X(g');
6      q_c := SplitLocation(A, C_Q, q, c);
7      add transition γ* = ⟨q_c, q', λ, g*, u⟩ to A;
8      if q = q' then
9          add transition ⟨q_c, q_c, λ, g, u⟩ to A;
10         add transition ⟨q_c, q_⊤, λ, g, u⟩ to A;
11     return γ*;
12 Function SplitLocation(A, C_Q, q, c) is
13     if q_c already exists in A then return q_c;
14     add a new location q_c to A;
15     C_Q(q_c) := c;
16     foreach incoming transition ⟨q̂, q, λ, g, u⟩ do
17         add transition ⟨q̂, q_c, λ, g, u⟩ to A;
18     foreach loop transition ⟨q, q, λ, g, u⟩ do
19         add transition ⟨q, q_c, λ, g, u⟩ to A;
20     if q is accepting then mark q_c as accepting;
21     if ∃ initial state (q, χ) with χ ⊨ c then add initial state (q_c, χ);
22     return q_c;
```

Algorithm 1: Well-constrained transformation for single transitions.

transition after the loop was split, the last loop iteration can be replaced with the copy made in Line 19.

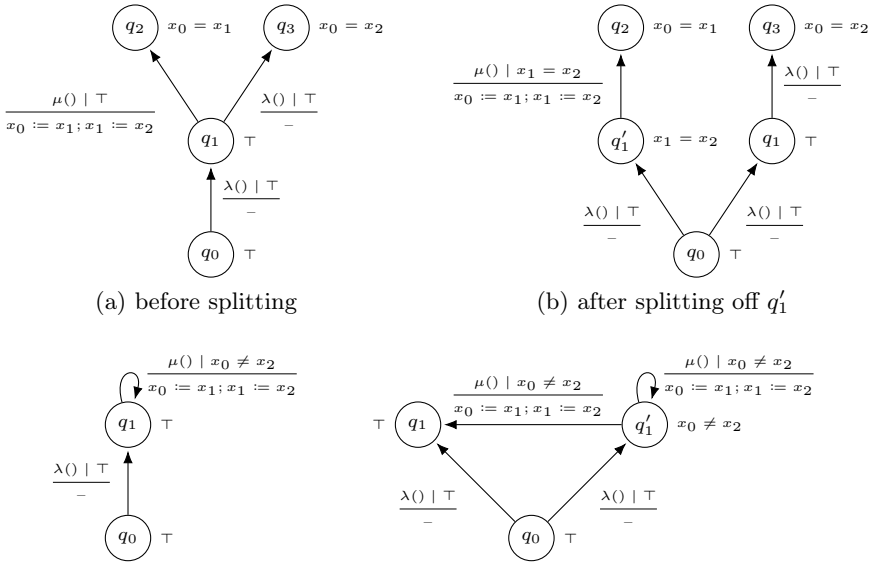
Incoming transitions For STS in A^*, replace the transition with the original variant. For STS in A, the replacement transition must be selected on the following transitions. If the transition is the last, select the transition to $q_\top$. If the next transition is γ, select the transition to the split-off location, if not, to the original.

Acceptance also remains identical due to Line 20. For initial locations, the argument is similar to outgoing transitions. If the first transition is γ, the original guard was satisfied and $\chi' \models C_Q(q')$, so the initial valuation $\chi \models C^*_Q(q_c)$ and q_c is initial. □

4.2 Iterative History Independence Transformation

To make an RA history independent, we can repeatedly apply Algorithm 1 until all transitions are history independent. This is formalized in Algorithm 2.

We will show the algorithm's correctness in multiple parts: first, we will demonstrate termination, then argue that the final set of location constraints is valid, and conclude that the resulting automaton must be history independent and its semantics are unchanged.



(a) before splitting (b) after splitting off q'_1

(c) before splitting (d) after splitting off q'_1

Fig. 4. Example results of the `MakeWellConstrained` function of Algorithm 1. (a) is transformed to (b) and (c) is transformed to (d).

Lemma 1. *Algorithm 2 terminates.*

Proof. The algorithm's state space is described by (Q_c, Γ_H), with Q_c being the constraint-specific locations and Γ_H the well-constrained transitions. We first define a lattice over the locations with their constraints. A location set Q_c is smaller than Q'_c if for every original location, the constraints of Q'_S have become stricter:

$$Q_c \sqsubseteq Q'_c \iff \forall q \in \text{original } Q \forall q_c \in Q_c \exists q'_c \in Q'_c : c \impliedby c'.$$

Note that $\mathsf{CC}[X]$ is finite. This allows us to define the state space as a lattice, with

$$(Q_c, \Gamma_H) \sqsubseteq (Q'_c, \Gamma'_H) \iff Q_c \sqsubseteq Q'_c \vee (Q_c = Q'_c \wedge \Gamma_H \subseteq \Gamma'_H).$$

Function `MakeHistoryIndependent`(A) **is**
  foreach *location* q **do** $C_Q(q) \coloneqq \top$;
  $\Gamma_H \coloneqq \emptyset$;
  while $\exists \gamma \notin \Gamma_H$ **do**
    $\gamma_H \coloneqq$ `MakeWellConstrained`(A, C_Q, γ);
    add γ_H to Γ_H;
  return (A, C_Q);

Algorithm 2: History Independence Transformation for RAs

In each step, the algorithm either marks a transition as well-constrained and leaves the automaton unchanged (Line 13), or splits a location and therefore strengthens its constraint. Therefore, it terminates in a finite number of steps due to the fixed point theorem of Knaster and Tarski [19, §2.1]. □

Lemma 2. *The location constraints returned by Algorithm 2 are valid for A.*

Proof. Since the algorithm terminates (Lemma 1), all transitions must be well-constrained transitions. For each initial STS, Line 21 ensures that the initial states satisfy their locations constraints. Inductively, by Theorem 4, if the nth states satisfies its constraint, so will the $n+1$st. Therefore, the constraints are valid for each reachable state. □

Theorem 6. *Algorithm 2 computes an history independent automaton accepting the same language as the input RA.*

Proof. Since Lemma 2 guarantees that all locations constraints are valid. All transitions are well-constrained, so by Theorem 3, they must be history independent. Because the initial constraints ($\top$) are valid for the input RA by definition, Theorem 5 guarantees that for each data word inducing an accepting initial STS in the original RA, an accepting initial STS in the history independent RA exists and vice versa, i.e., the accepted languages are identical. □

Finally, we show that the problem of transforming an RA into an equivalent history independent RA is FPSPACE-complete. This follows from NonEmptiness being PSPACE-complete in general, but simple to check on an history independent automaton.

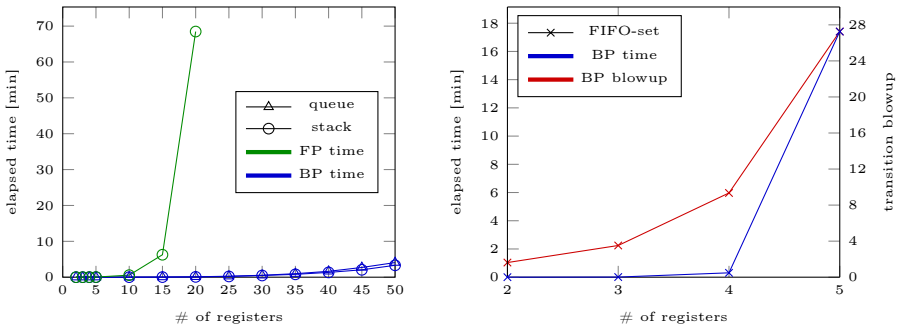
Corollary 1. *History independence is* FPSPACE*-complete.*

Proof Sketch. We begin by showing hardness by reducing the NonEmptiness problem on RAs. NonEmptiness for an arbitrary RA is PSPACE-complete, but NonEmptiness of a history independent register automaton can be tested by checking if an accepting location can be reached from an initial state's location in the transition graph. Since Reach on directed graphs in $\in$ NL, the history independence transformation must be at least FPSPACE-hard.

Now, we show that the transformation can be implemented using polynomial space. The state space of Algorithm 2 is bounded by the maximal number of transitions. For each original location, up to $3^{|X|^2}$ split-off versions may exist. The automaton size is at worst $|Q| \cdot 3^{|X|^2}$ (i.e., exponential in the original automaton's size). However, each state can be identified using a string of polynomial length, admitting an FPSPACE implementation. □

5 Evaluation

We implemented our back-propagating (BP) algorithms in Kotlin in the Koral software package and compared our performance to a symbolic *forward-*



(a) queues and stacks (b) FIFO-sets

Fig. 5. Evaluation results for data structure conformance checks. For all structures, the time required for successful forward- and back-propagation is shown, for FIFO-sets, the transition blowup caused by back-propagation is also rendered.

propagating (FP) conformance checker based on RALib[4] [8], JConstraints[5] [21] and Z3[6] [28].

As benchmarks, we used real-world-derived RAs provided in the Automata Wiki[7] [31] as well as random-generated register automata. For the Wiki automata, we simulated a conformance checking (CC) scenario by using the same automata as SUT and specification, i.e. we analyzed the self-product of each automaton.

We analyzed the performance of both our implementation and the RALib-based analyzer in the conformance checking scenario. Additionally, we studied our algorithm's performance on the original Wiki automata and the random RAs. For our algorithm, we recorded both the blowup (BU) in automaton size and the execution time. For the FP analyzer, only timing data was available. The experiments were executed on a Java VM with 32 GiB of heap memory running on an Intel Core i9-7960X CPU. A Docker image of the experimental setup is available on Zenodo [17].

5.1 Automata Wiki Benchmarks

The Automaton Wiki provides three sets of data structure benchmarks. These sets model bounded queues, stacks and FIFO-sets (stacks with a uniqueness constraint). The bound is determined by the number of registers; the wiki provides samples for up to 50 registers. The results of our evaluation are shown in Fig. 5. Forwards-propagation failed to analyze all queue and FIFO-set instances and all stacks with ≥ 25 registers. Below that, its performance is substantially worse than that of back-propagation. Since the automata contain no guards, BP only verifies

[4] https://bitbucket.org/learnlib/ralib/src/eqmc
[5] https://github.com/tudo-aqua/jconstraints
[6] https://github.com/Z3Prover/z3
[7] https://automata.cs.ru.nl

Table 1. Evaluation results on other real-world-derived automata. For all automata, the size, back-propagation performance on original and self-product (conformance check scenario) and forward-propagation performance on the product are shown.

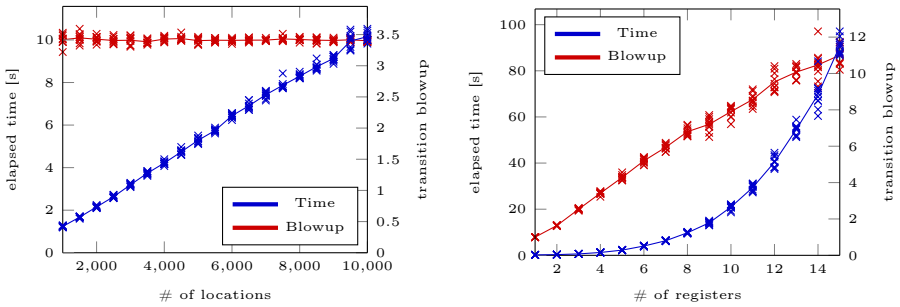
			BP Single			BP CC			FP CC
Automaton	$\lvert Q\rvert$	$\lvert\Gamma\rvert$	Q-BU	Γ-BU	Time	Q-BU	Γ-BU	Time	Time
ABP [4] Output[1]	30	50	1.00	1.00	0.08 s	0.80	0.92	0.20 s	DNF[3]
ABP [4] Receiver 3[1]	6	13	3.83	2.92	0.09 s	10.83	4.26	1.64 s	DNF[3]
ABP [4] Channel[1]	5	8	2.00	1.88	0.05 s	1.43	1.50	0.20 s	DNF[3]
FWGC[2] [32]	18	43	2.89	1.79	0.17 s	7.82	3.74	2293.08 s	0.84 s
Login [1]	12	20	1.25	1.15	0.04 s	2.50	2.23	0.12 s	0.55 s
Map	17	28	1.35	1.36	0.05 s	0.98	1.00	0.86 s	0.94 s
Overwriting Map	15	24	1.53	1.58	0.04 s	0.77	0.88	0.46 s	0.99 s
Passport[1] [3,22]	35	82	1.29	1.12	0.10 s	1.57	1.22	0.47 s	1.04 s
Repdigit Palindrome	6	23	4.00	1.78	0.04 s	10.33	1.92	0.18 s	0.45 s
SIP [1,2,34]	27	65	1.00	1.00	0.04 s	1.13	0.93	0.11 s	43.42 s

[1] automaton was manually transformed into our semantic [2] farmer, wolf, goat, and cabbage puzzle [3] execution failed with an error

that every transition is well-constrained and terminates without modifying the automaton. FIFO-sets demonstrate the limits of our analysis. These contain complex guards and our analysis times out after two hours for ≥ 10 registers and the analysis time increases sharply for five registers. The transition blowup (Γ-BU, i.e., the number of generated transitions divided by the number of original transitions) shows an immense increase in the automaton size caused by the transformation; the state blowup (Q-BU) is proportional and therefore not shown. The performance on non-product instances is comparable.

The results obtained for the remaining instances in the Automaton Wiki are shown in detail in Table 1. Two families of models that require more expressive semantics (randomness and arithmetic) were discarded. Some models were transformed to match our RA semantic without changing their input languages. While most instances were solved efficiently by both implementations, two edge cases stand out: backwards-propagating analysis is superior for the SIP implementation, while it exhibits extremely slow behavior (`MakeWellConstrained` is called 404 120 times) for the farmer, wolf, goat, and cabbage puzzle self-product without causing substantial blowup. The latter is due to the automaton having many registers, only two labels, and many loops, resulting in many location combinations in the product automaton which are not reachable from the initial states. Backwards propagation can only discover this by propagating location constraints along all possible paths instead of working "towards" a goal.

Other points of interest are that RALib failed to load the ABP models due to internal errors. Some models (e.g., APB Output and the map self-product) are history independent (blowup ≤ 1). Our algorithm occasionally produced a reduced automaton by removing dead locations or transitions from the product automata (blowup < 1).



(a) $|X| = 4$ & variable $|Q|$ (b) variable $|X|$ & $|Q| = 1000$

Fig. 6. Back-propagation evaluation results on random RAs. Arity is ≤ 4, 10 automata were generated per parameter set. The line follows the average results.

5.2 Random Automata

To evaluate the impact of several automaton properties, we implemented a random RA generator as part of Koral and ran our algorithm on families of generated automata. The generator offers three relevant tuneable parameters: maximum arity of symbols, number of registers and number of locations and transitions density. Preliminary analyses showed the impact of the maximum arity to be low (a minor increase in runtime), so we fixed the maximum arity at four. Figure 6 shows the result of the two remaining analyses: increasing number of locations with four registers and increasing number of registers with 1000 locations. All other parameters are fixed. As expected, the runtime is linear in the number of locations with a constant blowup of ca. 3.5. This shows that while more propagation steps are performed, their complexity remains constant. For the number of registers, a polynomial increase in runtime and a roughly linear increase in blowup can be observed. Again, this is to be expected: the back-propagation step is in $\mathcal{O}(|X|^3)$ and more registers enable more possible location constraints.

5.3 Threats to Validity

Since the algorithm is deterministic, all results for the automaton structure are fully replicable. In addition, our timing results were stable for multiple repetitions of the experiments. The performance on random automata is also stable and exhibits low variance. We therefore conclude that the results are internally valid.

However, while the Automaton Wiki automata are representative of real-world problems by design, only ten out of twelve families of automata could be analyzed by our tool. The random automata, while available in arbitrary number, are not necessarily representative of the real world. Therefore, the limited number of real-world automata available for study may limit external validity of our results.

6 Conclusion

Our algorithm provides a novel approach to NonEmptiness and Reach checking on register automata. The RA is first transformed into an equivalent, history independent RA. Then, Reach queries, including NonEmptiness, can be performed efficiently on the result. The performance of the combined process hinges on our backwards-propagation algorithm.

Compared to previous approaches, this algorithm provides a substantial increase in the number of automata that can be analyzed successfully. We showed that many real-world automata are history independent by design, and for many more, only a small penalty is incurred by the transformation. Our analysis also shows that there exist cases (e.g., the FIFO-set family) in which our approach, while better that prior ones, still performs badly. We additionally found a single case (FWGC self-product) in which our performance is substantially worse than that of previous approaches. Our experiments on random automata demonstrate that our algorithm can scale to very large automata. Its performance degenerates only in the presence of specific structures, not because of size alone.

We conclude that for most "interesting" real-world automata, the analysis speed and blowup do not reach exponential level, since their guard statements rarely need to be traced back multiple steps through a register automaton. Our approach is therefore suitable to analyze a larger subset of real-world register automata than previous approaches.

Future Work. Two independent directions of extension for our work are possible: extending the approach to more expressive languages and adapting the technique to more natural model checking approaches.

Our work can be naturally extended to languages over ordered fields such as $\mathbb{Q}$ and $\mathbb{R}$ with guards containing inequality operators. Recognizing languages over non-field structures is a more challenging task, since e.g. $(x < y) \wedge (y < z)$ can not be statically checked for satisfiability (consider $x = 1, z = 2$). Corresponding models were studied in, e.g., [7, 11]. Finally, an approach for register automata with a stack [29, 30] may be possible, but would require a backwards-propagation-compatible symbolic representation of stack states. When adding arithmetic operations to register automata, the resulting model becomes a Turing-complete counter automaton [26]. By adapting techniques such as bounded model checking, transformation of some of these automata into a history independent representation would still be possible.

Defining test automata to perform model checking is cumbersome and creating product automata results in a doubling of the number of registers and a corresponding performance penalty. We intend to explore the use of our approach to assist in the efficient evaluation of queries in logical languages such as LTL to offer a more natural and possibly performant approach to model checking.

References

1. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods. LNCS, vol. 7436, pp. 10–27. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_4
2. Aarts, F., Jonsson, B., Uijen, J.: Generating models of infinite-state communication protocols using regular inference with abstraction. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) Testing Software and Systems. LNCS, vol. 6435, pp. 188–204. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16573-3_14
3. Aarts, F., Schmaltz, J., Vaandrager, F.: Inference and abstraction of the biometric passport. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation. LNCS, vol. 6415, pp. 673–686. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16558-0_54
4. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A note on reliable full-duplex transmission over half-duplex links. Commun. ACM **12**(5), 260–261 (May 1969). https://doi.org/10.1145/362946.362970
5. Boigelot, B.: Symbolic methods for exploring infinite state spaces. Ph.D. thesis, Université de Liège, Liège, Belgium (May 1998), https://hdl.handle.net/2268/74874
6. Bojańczyk, M., Klin, B., Lasota, S.: Automata theory in nominal sets. Log. Methods Comput. Sci. **10**(4), 1–44 (Aug 2014). https://doi.org/10.2168/LMCS-10(3:4)2014
7. Brütsch, B., Landwehr, P., Thomas, W.: $\mathbb{N}$-memory automata over the alphabet $\mathbb{N}$. In: Drewes, F., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications. LNCS, vol. 10168, pp. 91–102. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-53733-7_6
8. Cassel, S., Howar, F., Jonsson, B.: RALib: a LearnLib extension for inferring EFSMs. In: Proceedings of the 4$^{\text{th}}$ International Workshop on Design and Implementation of Formal Tools and Systems (2015), https://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf
9. Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B.: A succinct canonical register automaton model. In: Bultan, T., Hsiung, P.A. (eds.) Automated Technology for Verification and Analysis. LNCS, vol. 6996, pp. 366–380. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_26
10. Cassel, S., Jonsson, B., Howar, F., Steffen, B.: A succinct canonical register automaton model for data domains with binary relations. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis. LNCS, vol. 7561, pp. 57–71. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_6
11. Chen, Y.F., Lengál, O., Tan, T., Wu, Z.: Register automata with linear arithmetic. In: 2017 32$^{\text{nd}}$ Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–12. IEEE (Jun 2017). https://doi.org/10.1109/LICS.2017.8005111
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: All-pairs shortest paths, chap. 25, pp. 684–707. MIT Press, Cambridge, third edn. (Feb 2009)
13. Czyba, C., Spinrath, C., Thomas, W.: Finite automata over infinite alphabets: two models with transitions for local change. In: Potapov, I. (ed.) Developments in Language Theory. LNCS, vol. 9168, pp. 203–214. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-21500-6_16

14. D'Antoni, L., Ferreira, T., Sammartino, M., Silva, A.: Symbolic register automata. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification. LNCS, vol. 11561, pp. 3–21. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_1
15. Demri, S., Lazić, R.: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. **10**(3), 16:1–16:30 (Apr 2009). https://doi.org/10.1145/1507244.1507246
16. Dierl, S., Howar, F.: A taxonomy and reductions for common register automata formalisms. In: Olderog, E.R., Steffen, B., Yi, W. (eds.) Model Checking, Synthesis, and Learning: Essays Dedicated to Bengt Jonsson on The Occasion of His 60$^{\text{th}}$ Birthday, LNCS, vol. 13030, pp. 186–218. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-91384-7_10
17. Dierl, S., Howar, F.: Reach on register automata via history independence – replication artifact (Mar 2022). https://doi.org/10.5281/zenodo.6367981
18. Garhewal, B., Vaandrager, F., Howar, F., Schrijvers, T., Lenaerts, T., Smits, R.: Grey-box learning of register automata. In: Dongol, B., Troubitsyna, E. (eds.) Integrated Formal Methods. LNCS, vol. 12546, pp. 22–40. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-63461-2_2
19. Granas, A., Dugundji, J.: Elementary fixed point theorems, chap. 2, pp. 9–84. Springer New York, New York, NY (2003). https://doi.org/10.1007/978-0-387-21593-8_2
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259
21. Howar, F., Jabbour, F., Mues, M.: JConstraints: A library for working with logic expressions in Java. In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not?, LNCS, vol. 11200, pp. 310–325. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-22348-9_19
22. International Civil Aviation Organization: Machine readable travel documents. Doc 9303, International Civil Aviation Organization, Montréal, Québec, Canada (2021), https://www.icao.int/publications/pages/publication.aspx?docnum=9303
23. Iosif, R., Xu, X.: Abstraction refinement for emptiness checking of alternating data automata. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10806, pp. 93–111. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_6
24. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994). https://doi.org/10.1016/0304-3975(94)90242-9
25. Madelaine, E., Qin, X., Zhang, M., Bliudze, S.: Using SMT engine to generate symbolic automata. Electron. Commun. EASST **76** (2019). https://doi.org/10.14279/tuj.eceasst.76.1103
26. Minsky, M.L.: Computation: Finite and infinite machines. Prentice-Hall International, London (1972)
27. Moerman, J., Sammartino, M.: Residual nominal automata. In: Konnov, I., Kovács, L. (eds.) 31$^{\text{st}}$ International Conference on Concurrency Theory. LIPIcs, vol. 171, pp. 44:1–44:21. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.CONCUR.2020.44
28. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

29. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Reachability in pushdown register automata. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) Mathematical Foundations of Computer Science 2014. LNCS, vol. 8634, pp. 464–473. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44522-8_39
30. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Reachability in pushdown register automata. J. Comput. Syst. Sci **87**, 58–83 (2017). https://doi.org/10.1016/j.jcss.2017.02.008
31. Neider, D., Smetsers, R., Vaandrager, F., Kuppens, H.: Benchmarks for automata learning and conformance testing. In: Margaria, T., Graf, S., Larsen, K.G. (eds.) Models, Mindsets, Meta: The What, the How, and the Why Not?, LNCS, vol. 11200, pp. 390–416. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-22348-9_23
32. Pressman, I., Singmaster, D.: The jealous husbands and the missionaries and cannibals. Math. Gaz. **73**(464), 73–81 (1989). https://doi.org/10.2307/3619658
33. Qin, X., Bliudze, S., Madelaine, E., Hou, Z., Deng, Y., Zhang, M.: SMT-based generation of symbolic automata. Acta Inform. **57**(3), 627–656 (Oct 2020). https://doi.org/10.1007/s00236-020-00367-6
34. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session initiation protocol. RFC 3261, RFC Editor (Jun 2002). https://doi.org/10.17487/RFC3261
35. Saint-Andre, P.: Extensible messaging and presence protocol (XMPP): Core. RFC 6120, RFC Editor (Mar 2011). https://doi.org/10.17487/RFC6120
36. Sakamoto, H., Ikeda, D.: Intractability of decision problems for finite-memory automata. Theor. Comput. Sci. **231**(2), 297–308 (2000). https://doi.org/10.1016/S0304-3975(99)00105-X