An Automata-Theoretic Approach to Reasoning About Parameterized Systems and Specifications

Orna Grumberg[1], Orna Kupferman[2], and Sarai Sheinvald[2]

[1] Department of Computer Science, The Technion, Haifa 32000, Israel
[2] School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel

Abstract. We introduce *generalized register automata* (GRAs) and study their properties and applications in reasoning about systems and specifications over infinite domains. We show that GRAs can capture both *VLTL* – a logic that extends LTL with variables over infinite domains, and *abstract systems* – finite state systems whose atomic propositions are parameterized by variable over infinite domains. VLTL and abstract systems naturally model and specify infinite-state systems in which the source of infinity is the data domain (c.f., range of processes id, context of messages). Thus, GRAs suggest an automata-theoretic approach for reasoning about such systems. We demonstrate the usefulness of the approach by pushing forward the known border of decidability for the model-checking problem in this setting. From a theoretical point of view, GRAs extend register automata and are related to other formalisms for defining languages over infinite alphabets.

1 Introduction

In model checking, we verify that a system has a desired behavior by checking that a mathematical model of the system satisfies a formal specification of the behavior. Traditionally, the system is modeled by a Kripke structure – a finite-state system whose states are labeled by a finite set of atomic propositions. The specification is a temporal-logic formula over the same set of atomic propositions [3].

When the system is defined over a large data domain or contains many components, its size becomes large or even infinite, and model checking may become intractable. Moreover, standard temporal logic may not be sufficiently expressive for specifying properties of such systems.

In [7], we introduced a novel approach for model checking systems and specifications that suffer from the size problem described above. Our approach extended both the specification formalism and the system model with atomic propositions that are parameterized by variables ranging over some (possibly infinite) domain. We studied the model-checking problem in this setting. While we showed that model checking in the general case is undecidable, we have managed to find interesting fragments of our systems and specification formalisms for which model checking is decidable. Our methods were based on reducing the problem to standard LTL model checking. The reduction was found helpful in some cases, but lacks a rigorous theoretical treatment. In particular, [7] left open the challenge of developing an automata-theoretic approach for this setting.

In this paper we introduce *generalized register automata* (GRAs), a new formalism for defining languages over infinite alphabets. GRAs can naturally model both the systems and specifications of [7]. We define GRAs, study their properties, and show how they not only provide a unified theoretical basis to the results in [7], but also enable strengthening and extending the results there.

We first elaborate on the setting in [7]. In an *abstract system*, our extension of a Kripke structure, every state is labeled by a set of atomic propositions. Some of the atomic propositions may be parameterized by variables that range over an unbounded or an infinite domain. The system also contains constraints on the possible assignments to the variables, and may reset their value during its execution. The *concrete computations* of an abstract system are induced by paths of the abstract system in which variables are assigned concrete values in a way consistent with the constraints and the resets along the path. For instance, if a path of the abstract system starts with $\{send.x\}, \{rec.x\}, \{send.x\}$, and x is a variable over $\mathbb{N}$ that is reset between the second and third state, then a concrete computation induced by the path may start with $\{send.3\}, \{rec.3\}, \{send.5\}$. Evidently, abstract systems are capable of describing communication protocols with unboundedly many processes, systems with interleaved transactions each carrying a unique id, buffers of messages with an infinite domain, and many more.

Our specification formalism, *Variable LTL* (VLTLs), also uses atomic propositions parameterized by variables. For example, the VLTL formula $\forall x.\boldsymbol{G}\,(send.x \rightarrow \boldsymbol{F}\, receive.x)$ states that for every value d in the domain, whenever a message with content d is sent, then a message with content d is eventually received. As another example, the formula $\exists x.\boldsymbol{G}\,\boldsymbol{F}\,\neg idle.x \wedge \boldsymbol{G}\,\boldsymbol{F}\,\neg idle.x$ states that in each computation, there exists at least one process that is both idle and not idle infinitely often. Note that if the domain of messages or process id's is infinite or unknown in advance, then there exist no equivalent LTL formulas for these VLTL formulas.

As described above, in [7] we solved the VLTL model-checking problem for some fragments of the (undecidable) general setting. Our goal here is to suggest an automata-theoretic approach to the problem, hopefully pushing the boundaries of decidable fragments. In the automata-theoretic approach to model checking [15], we represent systems and their specifications by automata on infinite words. Questions such as model checking and satisfiability are then reduced to questions about automata and their languages. Traditional automata are too weak for modeling abstract systems or VLTL formulas, and a formalism that can handle infinite alphabets is needed.

A classical formalism for defining languages with an infinite alphabet is that of *register automata* [8, 9]. A nondeterministic register automaton comprises a state machine and a finite set of registers that may store values of the infinite domain. In a transition, the register automaton either guesses some value and stores it in one of the registers (an ϵ-transition), or advances on the input word if the content of register in the transition matches the next input letter.

Our formalism of GRA extends register automata in a way that enables easy modeling of abstract systems and VLTL formulas.[3] Essentially, this involves features that mimic the conjunctions and disjunctions in the logic (that is, the transition function of GRAs is *alternating*), features that mimic the existential and universal quantification of variables (that is, GRAs have two types of ϵ-transitions, one – $\epsilon_\exists$, which guesses and assigns a single value to a register, and one – $\epsilon_\forall$, which assigns all possible values to a register by splitting the run, creating a different copy for every such value), features that mimic the constraints on the variable values (by adding constraints on the content of the registers), and features that make it possible to complement a given GRA by dualization (by closing the components of a transition, namely the branching mode and guards, to dualization).

We formally define GRAs and study their theoretical properties. We show that GRAs are closed under the Boolean operations. Unsurprisingly, their universality and emptiness problems are generally undecidable, yet we point to the fragment in which the GRAs have only a single register, for which nonemptiness is decidable.

We compare GRAs with the formalisms of register automata and data automata [9, 2]. We show that GRAs are strictly more expressive than register automata. We describe a translation from deterministic data automata to GRAs and show that there are languages that are accepted by GRAs and not by (nondeterministic) data automata.

We describe a translation of abstract systems and VLTL formulas to GRAs. The translation of a VLTL formula to an equivalent GRA resembles the translation of LTL formulas to nondeterministic Büchi automata [15]. The quantifiers in the formula are handled by a sequence of $\epsilon_\exists$ (for $\exists$ quantifiers) and $\epsilon_\forall$ transitions (for $\forall$ quantifiers).

In [7], we showed that model checking is undecidable already for VLTL formulas with two $\exists$ quantifiers, and is decidable for formulas with only $\forall$ quantifiers. The translation to GRA enables us to complete the picture and show that for the safe fragment of VLTL, model checking of formulas of type $\forall x_1; \forall x_2; \ldots \forall x_k; \exists x \varphi$ is decidable. This is a useful fragment, as it captures specifications of the form "for every environment, there exists a value that satisfies φ". As an example, consider the formula $\forall x_1; \exists x_2; \mathbf{G}\left((\neg idle.x_1) \rightarrow \mathbf{X}\left(\neg idle.x_2\right)\right)$, with $x_1 \neq x_2$. This formula states that if there exists some non-idle process, then it will be immediately followed by a different non-idle process, thus ensuring that there is an infinite sequence of non-idle processes. Another example is the formula $\forall x_1; \exists x_2; \mathbf{G}\left(\left((\neg req.x_1) \wedge \mathbf{X} req.x_1\right) \rightarrow \mathbf{X} new\_process.x_2\right)$, stating that whenever a request with new content is sent, a new process with a new process id is envoked. Dually, the satisfiability of formulas of the type $\exists x_1; \exists x_2; \ldots \exists x_k; \forall x \varphi$ is also decidable. For formulas of the type $\exists x_1; \forall x_2; \varphi$, model checking is again undecidable.

Our upper-bound proofs rely on a reduction to the nonemptiness problem for multi-counter machines. The model-checking complexity in these cases is then non-elementary. Finding a lower bound has the same flavor as finding a lower bound for the nonemptiness of data automata [2], which also uses multi-counter machines to show the decidability of nonemptiness, and is a problem that is still open.

[3] We study GRA on finite words. Extending the definition to infinite words is easy and the technical difficulties are orthogonal to these that the setting of infinite alphabets involves. Thus, the results here are restricted to the safe fragment of VLTL.

Related Work There are quite a few different models and variants of automata over infinite alphabets, differing in their expressive power and decidable properties. A major motivation for such models origins from formal reasoning about XML [13].

Register automata were first introduced in [8]. These were extended in [9] to include ϵ-transitions. In [6], we studied VFA, a sub-type of nondeterministic register automata that can be represented by finite automata and has fragments that are closed under the Boolean operations.

Several types of *alternating register automata* (ARA) have been studied, differing in their expressive power. In [12], the state machine has universal and existential states. The run on a universal state splits into all possible configurations that may follow the current configuration. [12] also studies the two-way model. In [5], the automaton is single-register, and is enriched with the actions $guess$ (an ϵ-transition) and $spread$ (creating new threads of the run with all data values that appear with some state, starting from another state). For this model, nonemptiness is decidable. In [4], the authors study the relations between LTL with the freeze quantifier (an extension of LTL that is equiped with a register) and single-register alternating register automata.

Another type of automata over infinite alphabets are *data automata* [2]. Data automata are defined over alphabets of the type $\Sigma \times D$, where Σ is finite and D is infinite. Intuitively, Σ is accessed directly, while D can only be tested for equality, and is used for inducing an equivalence relation on the set of positions. Technically, a data automaton consists of two components. The first is a letter-to-letter transducer that runs on the projection of the input word on Σ and generates words over yet another alphabet Γ. The second is a finite automaton that runs on subwords (determined by the equivalence classes) of the word over Γ generated by the transducer. Data automata turn out to be a very expressive model for which nonemptiness is decidable (albeit non-elementary). [10] and [17] study weaker versions of data automata, for which nonemptiness is elementary.

Data automata too have several extensions. Such an extension is *class automata* [1], which were defined for the purpose of studying of XPath. A class automaton behaves almost similarly to a data automaton, but the automaton component processes the entire word that is produced by the transducer (as opposed to processing a subword of it), and it takes special transitions when it reads letters of the class it handles. This modification makes nonemptiness undecidable for this type. Other models limit the structure of the automaton component of class automata [16], or add counters to the different data values [11] to achieve decidable emptiness.

A third type are *pebble automata* and their variants. A pebble automaton [12] places pebbles on the input word in a stack-like manner. The transitions of a pebble automaton compare the letter in the input with the letters in positions marked by the pebbles. Several variants of this model have been studied. For example, [12] studies alternating and two-way pebble automata, and [14] introduces top-view weak pebble automata.

These formalisms are insufficient for our purposes of studying of VLTL and abstract systems – both in terms of expressive power, and in terms of easiness of translation. Our formalism of GRA is designed specifically to deal with this setting, and offers a clean and natural translation and suitable decidable fragments.

2 Preliminaries

Automata on Data Words *Data words* are words over an infinite alphabet $\Sigma \times D$, where Σ is a finite set to which we refer to as *labels*, and D is an infinite set to which we refer to as *data*.

A *nondeterministic register automaton on data words (NRA)* $\mathcal{A}$ comprises an alphabet $\Sigma \times D$, a set $r = \{r_1, r_2, \ldots r_k\}$ of registers that can contain a value of D each, an initial register assignment $r_\# \in (D \cup \{\#\})^k$ where $\# \notin D$, a set of states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, and a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times [k] \times Q$, where $[k] = \{1, 2, \ldots, k\}$.

A run on an input word w over $\Sigma \times D$ begins at state q_0, and r_i is assigned $r_\#(i)$ for $1 \leq i \leq k$. Intuitively, when $\mathcal{A}$ is in state q and the next input letter is $\langle a, d \rangle$, if it takes a transition labeled $\langle \epsilon, i \rangle$, then it nondeterministically stores some value in register r_i that is different from the contents of the rest of the registers, and does not advance on the input word. A transition labeled $\langle a, i \rangle$ may be taken if the content of the register r_i is d, in which case $\mathcal{A}$ also advances to the next input letter.

The word w is accepted by $\mathcal{A}$ if there exists a run on w that advances along all of w and reaches an accepting state. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all words accepted by $\mathcal{A}$.

Data automata [2] are another formalism that handles data words. A data automaton $\mathcal{C}$ is a tuple $\langle \Sigma \times D, \Gamma, A, B \rangle$, where A is a letter-to-letter transducer whose input alphabet is Σ and output alphabet is Γ, and B is an NFA over Γ.

To explain the way a data automaton operates, we begin with some terms and notations. Consider a word $w = \langle a_1, d_1 \rangle \langle a_2, d_2 \rangle \ldots \langle a_n, d_n \rangle$ over $\Sigma \times D$. The *string projection* of w is the word $a_1 a_2 \ldots a_n$. A *class* in w is a maximal set of indices for which the letters in w in these indices share the same data value. For example, in the data word $\langle a, 1 \rangle \langle b, 1 \rangle \langle b, 2 \rangle \langle c, 1 \rangle \langle a, 2 \rangle$, there are two different classes: $\{1, 2, 4\}$ and $\{3, 5\}$. Every class induces a *class word*, a word over Σ that is formed by concatenating the labels of the matching letters of the class in the order in which they appear in w. In the example, the two class words are abc and ba.

Consider a word $w = \langle a_1, d_1 \rangle \langle a_2, d_2 \rangle \ldots \langle a_n, d_n \rangle$ over $\Sigma \times D$. A run of $\mathcal{C}$ on w consists of two parts. First, the transducer A runs on the string projection of w and outputs a word $\gamma_1 \gamma_2 \ldots \gamma_n$ over Γ. If it rejects then the run is rejecting. Otherwise, the automaton B runs on every class word of $\langle \gamma_1, d_1 \rangle \langle \gamma_2, d_2 \rangle \ldots \langle \gamma_n, d_n \rangle$. If B accepts all the class words then the run is accepting, otherwise it is rejecting.

The class of data automata contains the class of register automata, and the emptiness problem for data automata is decidable. However, data automata are not closed under complementation [2].

Abstract Systems and VLTL In [7], we introduced *variable LTL* (VLTL) and *abstract systems*. For both, the standard formalism of Kripke structures and LTL formulas is extended with a set of variables that enables the computations of the Kripke structure to carry values over some infinite domain D, and the formulas to express properties with respect to these values. More specifically, the standard finite set of atomic propositions AP over which both the systems and the formulas are defined is extended by a finite

set of *parameterized atomic propositions* T. The propositions of T are parameterized by variables from a finite set X. These variables are assigned values from D.

An abstract system S is a finite Kripke structure over $AP \cup (T \times X)$. In every transition of S, a subset X' of X may be reset, meaning that the varibles of X' may change their value in the next step. The system S also includes an inequality set E over X. Having $x_i \neq x_j \in E$ means that in every point of the computation, the value assigned to x_i must be different from the value that is assigned to x_j. It holds that for every system S there exists an equivalent system S' over the same set of variables such that the inequality set S' is the full inequality set $\{x_i \neq x_j | x_i, x_j \in X\}$. A *computation* π of S is then an infinite word over $2^{AP \cup (T \times D)}$, induced by some infinite path w (over $2^{AP \cup (T \times X)}$) of S. The D values in π_i are obtained by the assignment to the variables in w_i. These values comply both with E and with the resets that w traverses – the value of a variable does not change as long as it has not been reset.

A VLTL formula is a pair $\langle \varphi, E \rangle$, where $\varphi = Q_1 x_1; Q_2 x_2; \ldots Q_k x_k; \psi$, where $Q_i \in \{\forall, \exists\}$ and x_i is a variable in X for every $1 \leq i \leq k$, where ψ is an LTL formula over $AP \cup (T \times X)$, and E is an inequality set over the variables. The semantics of VLTL is with respect to computations over $2^{AP \cup (T \times D)}$ and assignments to the variables of φ. Intuitively, a computation π satisfies a formula $\exists x; \psi$ (denoted $\pi \models \exists x; \psi$) if there exists some value d that may (w.r.t. E) be assigned to x such that $\pi \models \psi[x \leftarrow d]$ in the LTL sense. Similarly, π satisfies $\forall x; \psi$ if for every value d that may be assigned to x, it holds that $\pi \models \psi[x \leftarrow d]$. For the formal definition, see [7].

We say that a system S *satisfies* a VLTL formula $\langle \varphi, E \rangle$ (denoted $S \models \langle \varphi, E \rangle$), if every computation of S satisfies $\langle \varphi, E \rangle$. The model-checking problem for VLTL and abstract systems is then to decide, given S and $\langle \varphi, E \rangle$, whether $S \models \langle \varphi, E \rangle$. In [7], we showed that this problem is generally undecidable, already for formulas of the type $\exists x_1; \exists x_2; \psi$, where ψ is quantifier free. We showed, however, that model checking is decidable when there are no resets in the system. Further, model checking is decidable also in the case where the VLTL formula contains only $\forall$ quantifiers.

3 Generalized Register Automata

We present a generalization of register automata, called *generalized register automata* (GRA), that allows alternation and dualization of the conditions on the transition. The following details are generalized.

- Recall that in an $\langle \epsilon, i \rangle$ transition, if the automaton stores some value in register r_i, then it must be different from the values in all other registers. We generalize this idea by labeling every ϵ-transition by a Boolean formula over inequalities between the registers (to which we also refer as a *guard*). For the run to continue along an ϵ transition, the register assignment must satisfy the guard condition.
- Recall that in an $\langle \epsilon, i \rangle$ transition, the automaton nondeterministically stores some value in register r_i. We can view this as follows: The run is accepting if there exists some value that is stored in r_i, such that the rest of the run is accepting. We generalize this notion by defining two types of ϵ-transitions: in an $\langle \epsilon_\exists, i \rangle$-transition, the run is accepting if there exists some legal (w.r.t. the guard condition) value that

is stored in r_i, such that the rest of the run is accepting. in an $\langle \epsilon_\forall, i \rangle$ transition, the run is accepting if for every value that is stored in r_i, the rest of the run is accepting.
- In the definition of register automaton, the state machine component is nondeterministic. We generalize this by allowing the state machine to be alternating.

Formally, a *generalized register automaton* (GRA) is a tuple

$$\langle \Sigma \times D, \#, r, r_\#, Q, q_0, \delta, F \rangle,$$

where

- $\Sigma \times D$ is the input alphabet, where Σ is finite and D is infinite,
- $r = \{r(1), r(2), \ldots r(k)\}$ is a finite set of registers,
- $\# \notin \Sigma$ marks an empty register,
- $r_\# \in (\Sigma \cup \{\#\})^k$ is the initial register assignment,
- Q is a finite set of states,
- q_0 is the initial state,
- $F \subseteq Q$ is a set of accepting states, and
- $\delta \subseteq (Q \times \Sigma \times B^+(Q \times [k])) \cup (Q \times \{\epsilon_\exists, \epsilon_\forall\} \times B^+(Q \times G(r) \times [k]))$ where $G(r)$ is the set of guards over inequalities over $\{r(1), r(2), \ldots, r(k)\}$, and B^+ stands for the set of positive Boolean formulas [4].

We describe a run of a GRA on an input word w. Since GRAs are alternating, a run on w is a tree. Each node of the tree holds the following information: the current state, the current register configuration, and the current position in the input word. The root of the tree is labeled $\langle q_0, r_\#, 1 \rangle$.

The sons of a node x labeled $\langle q, \langle d(1) \ldots d(k) \rangle, i \rangle$ are determined by the type of transition that is taken from x: an ϵ-transition (an $\epsilon_\exists$-transition or $\epsilon_\forall$-transition), or a transition that advances on the input word. We describe how the run continues from x for each of these transitions.

In the case of an $\epsilon_\exists$ transition, suppose that $\langle q_1, g_1, k_1 \rangle, \langle q_2, g_2, k_2 \rangle, \ldots \langle q_p, g_p, k_p \rangle$ is a satisfying set for $\delta(q, \epsilon_\exists)$. Then from x, the run can continue by splitting into the son nodes $x_1, x_2, \ldots x_p$. These sons are all located in position i in w (that is, they do not advance on the input word). A branch that leads from x to a son x_j assigns a value to register k_j in a way that agrees with g_j, and moves to state q_j. Therefore, x_j is labeled $\langle q_j, \langle d_j(1), d_j(2) \ldots d_j(k) \rangle, i \rangle$, where $\langle d_j(1), d_j(2) \ldots d_j(k) \rangle$ satisfies g_j, and may differ from $\langle d(1) \ldots d(k) \rangle$ only in the register k_j.

In the case of an $\epsilon_\forall$ transition, again suppose that $\langle q_1, g_1, k_1 \rangle, \ldots \langle q_p, g_p, k_p \rangle$ is a satisfying set for $\delta(q, \epsilon_\forall)$. Then from x, the run continues by splitting into infinitely many son nodes, all located in position i on w. For every $\langle q_j, g_j, k_j \rangle$, the run branches over all values that can be stored in register k_j and satisfy g_j. Thus, for every $\langle q_j, g_j, k_j \rangle$, for every value d that can be stored in register k_j in a way that satisfies g_j, the node x has a son labeled q_j, in position i, whose register assignment is identical to that of x, except for register k_j, which stores the value d.

[4] The full definition of GRA also includes the classification of the transitions to "may" and "must" transitions, which allows easy dualization and complementation. For simplicity, and since we do not use these features in our results, we omit them from the definition.

Finally, for a transition that advances on the input word, suppose that $\langle q_1, k_1\rangle, \ldots$ $\langle q_p, k_p\rangle$ is a satisfying set for $\delta(q, \sigma)$, and that $w_i = \langle a, d\rangle$. Then from x, the run can continue by splitting into the son nodes $x_1, x_2, \ldots x_p$. These sons are all located in position $i+1$ in w (that is, they advance one letter on the input word), and their register configuration is identical to the register configuration of x. A branch that leads from x to a son x_j must hold the value d in its register k_j.

In the representation of δ, the registers to be read or written to are paired with the state the transition leads to. This is essential for alternation. However, to make δ more convenient to read, for the rest of the paper we represent it similarly to the transition of NRA whenever possible. Also, in most cases we discuss the guards are uniform throughout the GRA, and so we omit the guards from the representation, noting them elsewhere. For example, we represent a transition $\langle q, \epsilon_\exists, \langle s, r_1 \neq r_2, 1\rangle \vee \langle t, r_1 \neq r_2, 2\rangle\rangle$, as two transitions from q; one labeled $\langle \epsilon_\exists, 1\rangle$ leading to s, and one labeled $\langle \epsilon_\exists, 2\rangle$ leading to t. Similarly, we represent a transition $\langle q, a, \langle s, 1\rangle\rangle$ as a transition from q labeled $\langle a, 1\rangle$, leading to s.

Example 1. Figure 1 displays the three types of transitions. In (a), an $\epsilon_\exists$ transition is followed from q with a satisfying set $\langle s, 2\rangle, \langle t, 2\rangle$, and $\langle t, 3\rangle$ (we omit the guard conditions, that state that the assignment to all registers must be different). The tree branches accordingly: the leftmost and middle sons are in states s and t, respectively, reassigning the second register (the middle son reassigns it with the same value it held before), and the rightmost son is in state t and reassigns the third register.

In (b), an $\epsilon_\forall$ transition is followed from q with a satisfying set $\langle s, 2\rangle$, and again we omit the guard condition. Then the run branches into all possible assignments to the second register, in each path moving to state s.

In (c), the input letter $\langle a, 4\rangle$ is read on a transition from q with a satisfying set $\langle s, 1\rangle$ and $\langle t, 2\rangle$. Then the run splits to two son nodes s and t, where the path to s reads the value 4 from the first register, and the path to t reads 4 from the second register.

For convenience, we label the edges by the transitions, represented as a transition for NRA, as we have explained above.

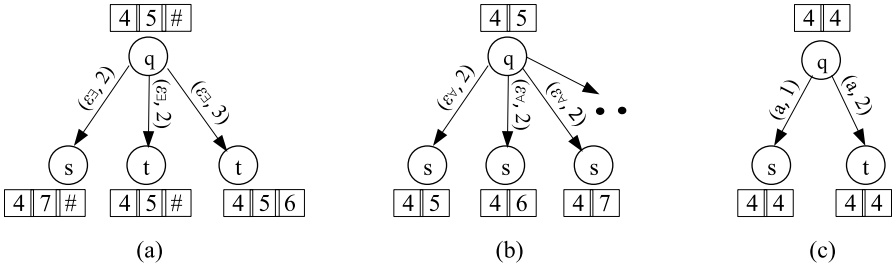

Fig. 1. The three types of transitions (a), (b) and (c).

The run tree is *accepting* if the leaves of all paths of the tree that have read all of w are in an accepting state. Notice that the ϵ-transitions may result in infinite paths that do not advance on the word. Since the definition of acceptance only considers the leaves, these paths are ignored when deciding acceptance. A word w is accepted by a GRA $\mathcal{A}$ if $\mathcal{A}$ has an accepting run tree on w [5].

Example 2. Consider the GRA $\mathcal{A}$ seen in Figure 2. In every transition of $\mathcal{A}$, the guard is $r_1 \neq r_2$, and we omit this detail from Figure 2 for the easiness of reading. The language of $\mathcal{A}$ is the set of all nonempty words over $\{a\} \times D$ in which no data value is repeated. Every run of $\mathcal{A}$ first splits over all values stored in r_1. Then, in every copy, as long as the next input value is different from r_1, the run continues by storing and reading the next value in r_2. The value in r_1 may only be read once and then cannot be read again from state s. Notice that all copies that do not have a value of the input word in their r_1 stay and accept from state r. Figure 2 also includes an accepting run tree on the word $\langle a, 3\rangle\langle a, 4\rangle$.

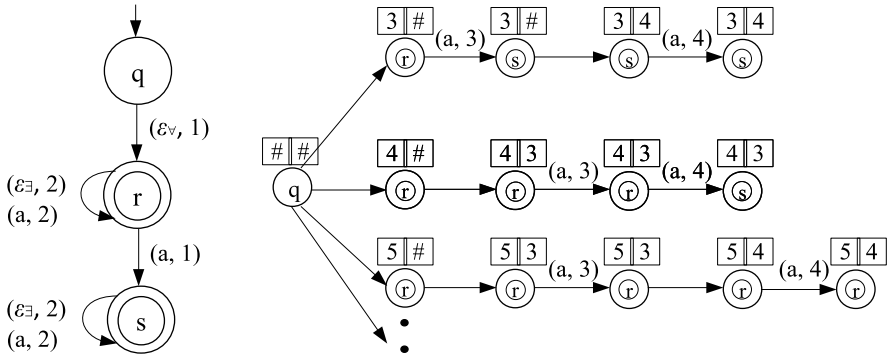

Fig. 2. The GRA $\mathcal{A}$ and an accepting run on the word $\langle a, 3\rangle\langle a, 4\rangle$.

Since GRAs are a generalization of nondeterministic register automata, we have that every NRA has an equivalent GRA.

Given two GRAs $\mathcal{A}$ and $\mathcal{B}$ with sets of registers $r_{\mathcal{A}}$ and $r_{\mathcal{B}}$, respectively, we can easily construct GRAs $\mathcal{A}\cup\mathcal{B}$ and $\mathcal{A}\cap\mathcal{B}$ for the languages $\mathcal{L}(\mathcal{A})\cup\mathcal{L}(\mathcal{B})$ and $\mathcal{L}(\mathcal{A})\cap\mathcal{L}(\mathcal{B})$, respectively, as follows. For both constructions, the set r of registers is a concatenation of $r_{\mathcal{A}}$ and $r_{\mathcal{B}}$, and the state machine is the union of the state machines of $\mathcal{A}$ and $\mathcal{B}$, with the addition of a single new initial state q_0. The transition for q_0 in $\mathcal{A} \cup \mathcal{B}$ and in $\mathcal{A} \cap \mathcal{B}$ is defined $\delta(q_0) = \delta_{\mathcal{A}}(q_0^{\mathcal{A}}) \vee \delta_{\mathcal{B}}(q_0^{\mathcal{B}})$ and $\delta(q_0) = \delta_{\mathcal{A}}(q_0^{\mathcal{A}}) \wedge \delta_{\mathcal{B}}(q_0^{\mathcal{B}})$, respectively, where

[5] We could define Büchi acceptance conditions for infinite words as well in the standard way, in which a run tree is accepting if all of its paths that infinitely often advance on the input word, infinitely often traverse some accepting state. As we have mentioned, in this paper we concentrate on finite words.

$q_0^{\mathcal{A}}$ and $\delta_{\mathcal{A}}$ are the initial state and transition function of $\mathcal{A}$, and similarly for $q_0^{\mathcal{B}}$ and $\delta_{\mathcal{B}}$. That is, the construction for the union and intersection is the standard construction for alternating automata, and the set of registers is obtained by simply concatenating the sets of registers for both automata.

We can reduce PCP (Post's correspondence problem) to both the universality problem and the nonemptiness problem for GRA, and so they are both undecidable. However, given a word w and a GRA $\mathcal{A}$, it is decidable whether $\mathcal{A}$ accepts w. To see why, notice that the precise identity of the data values that do not appear in w and are assigned to the registers during a run does not matter. What matters are only the equality relations between them. Then, we can show that a run tree of $\mathcal{A}$ on w can be simulated by using a bounded number of values (that depends on the the number of different values in w and the number of registers in $\mathcal{A}$), without using $\epsilon_\forall$-transitions. Further, we can also bound the number of consecutive ϵ-transitions in every path, and so it suffices to check trees of bounded width and bounded length to decide whether $w \in \mathcal{L}(\mathcal{A})$.

Finally, we can show that for the single-register fragment of GRA, the nonemptiness problem is decidable. The next theorem sums up the closure and decidability properties of GRA.

Theorem 1. *1. GRAs are closed under union and intersection.*
2. The membership problem for GRAs is decidable.
3. The nonemptiness and universality problems for GRAs are undecidable.
4. The nonemptiness problem for GRAs with a single register is decidable.

We now proceed to compare data automata to GRA. A deterministic data automaton $\mathcal{C}$ can be translated to an equivalent GRA with two registers r_1, r_2 as follows. Using an $\epsilon_\forall$-transition on r_1, the GRA splits into infinitely many copies. Each copy checks a different class of the input word, where the class is determined by its content of r_1, and simulates a simultaneous run on both the transducer and the automaton components of $\mathcal{C}$; upon reading a letter, if the data is not the class it needs to check, then the copy only advances on the transducer (using r_2 to guess and advance on this data). If the data is the content of r_1, then the copy advances along both the transducer and the automaton. The copy accepts if both the transducer and the automaton reach an accepting state.

In [2], the authors point to a language that cannot be accepted by a data automaton. This language can be accepted by a GRA with three counters. Roughly speaking, a GRA can accept languages of words of the form $w\#w$, and data automata cannot. Therefore, data automata are not stronger than GRA. We leave the precise comparison with data automata open.

Theorem 2. *1. Every deterministic data automaton has an equivalent GRA.*
2. Data automata are not more expressive than GRA.

4 From VLTL and Abstract Systems to GRA

In this section, we show how to translate VLTL formulas and abstract systems to GRAs. Then, we use these constructions to find fragments of VLTL for which the satisfiability and model checking problems are decidable.

Since GRAs are capable of expressing a single value in every letter, we cannot directly express computations in which more than one value appears at a time, and we first concentrate on a restricted type of computations that include a single value in every state. Then, we show how to encode unrestricted computations with restricted ones.

A computation π over $2^{AP \cup (T \times D)}$ is called *restricted* if π_i contains at most one data value for every i.

Let S be an abstract system with k variables and the full inequality set, in which every state contains at most one variable. It is easy to see that this is a sufficient and necessary condition for S to have only restricted computations. An equivalent GRA $\mathcal{A}_S$ is obtained from the structure of S by using k registers, where register r_i holds the data value assigned to the variable x_i. Resets are translated to $\epsilon_\exists$ transitions, and the inequality set is reflected in the guard conditions. All states of $\mathcal{A}_S$ are accepting. Clearly, $\mathcal{L}(\mathcal{A})$ is exactly the set of all concrete computations of S.

Given a VLTL formula $\langle \varphi = Q_1 x_1; Q_2 x_2; \ldots Q_k x_k; \psi, E \rangle$, where $Q_i \in \{\exists, \forall\}$ for every i, where E is an inequality set over the set of variables and where ψ is an LTL formula over $AP \cup (T \times X)$, we construct a GRA $\mathcal{A}_\varphi$ with $k+1$ registers over $2^{(AP \cup T)} \times D$, whose language is exactly the set of restricted computations that satisfy φ. A letter $\langle s, d \rangle$ represents a set of atomic propositions $s \in 2^{AP \cup T}$, such that the parameterized atomic propositions in s all carry the same value d.

For simplicity, we assume that E states that the value of all variables must be different. A general E can then be handled by the guards in the transitions, requiring that if a set of variables appears in a transition, then they all must carry the same value.

Intuitively, the construction of $\mathcal{A}_\varphi$ relies on the Vardi-Wolper construction for ψ. The variables are handled by a set of $k+1$ registers, and the quantifiers are translated to an $\epsilon_\exists$-transition for an $\exists$ quantifier, and to an $\epsilon_\forall$-transition for a $\forall$ quantifier.

Thus, the run begins by following a sequence of states and transitions matching the sequence of quantifiers in φ; for every $1 \leq i \leq k$, an occurence of $\exists x_i$ is translated to an $\langle \epsilon_\exists, i \rangle$-transition, and an occurence of $\forall x_i$ is translated to an $\langle \epsilon_\forall, i \rangle$-transition. The inequality set is reflected in the transitions within this sequence, that makes sure that the values stored in registers r_1 through r_k satisfy E. Since we assume that E is the full inequality set, this means that in every copy, every register contains a different value from the other registers. Once the values are stored, in every copy of the run, the registers r_1 through r_k are fixed, while register r_{k+1} handles values in the computation that do not appear in any of the registers.

The NRA component of $\mathcal{A}_\varphi$ then behaves as the automaton for ψ with the following changes.

- Every state may change the value of r_{k+1} to some value different from the values in the rest of the registers, by following self loops labeled $\langle \epsilon_\exists, k+1 \rangle$.
- Recall that we allow only a single value to appear in every step. However, in the Vardi-Wolper construction the labeling is over $2^{AP \cup (T \times X)}$. We therefore restrict the labels to those that contain a single variable of X.

  Now, consider a transition labeled by a set that contains no variables at all (that is, its set of atomic propositions is $A \subseteq AP$). This means that $a.x$ is set to false for every $a \in T$ and $x \in X$. This can hold if either a does not hold in this step for any value, or if a holds with a value that is different from the values that are assigned

to the variables in X. The register r_{k+1} may hold this value. Therefore, for every $B \subseteq T$, we add to this transition the label $\langle A \cup B, k+1 \rangle$.

The following theorem summarizes this discussion.

Theorem 3. *1. For every abstract system S with restricted computations there exists a GRA $\mathcal{A}_S$ such that $\mathcal{L}(\mathcal{A}_S)$ is the set of computations of S.*
2. For every VLTL formula φ there exists a GRA $\mathcal{A}_\varphi$ such that $\mathcal{L}(\mathcal{A}_\varphi)$ is the set of restricted computations that satisfy φ.

We handle unrestricted computations by encoding the content of a single state by a sequence of letters, each carrying a single value. The alphabet is $2^{AP} \cup (T \times D)$[6]. Intuitively, a letter of type $\langle t, d \rangle$ represents $t.d$ appearing in the state, and a letter in 2^{AP} represents the set of unparameterized atomic propositions in the state. A sequence that matches a state first lists the unparameterized atomic propositions, and then lists the parameterized atomic propositions, one by one. Thus, when translating an abstract system or a VLTL formula over $X = \{x_1, x_2, \ldots x_k\}$ to a GRA, each label is translated to a sequence of labels in $2^{AP} \cup (T \times [k])$ (or $2^{AP} \cup (T \times [k+1])$ for a VLTL formula).

For model checking purposes, we make sure that: (a) both the system GRA and the VLTL GRA have a uniform representation of each label, which is done listing the content of each state according to some predefined order $<$ on T, and (b) the VLTL GRA is reverse-deterministic (a property that is essential for the decidability of the construction in Theorem 6 below). This is achieved by changing the alphabet of the VLTL GRA to $2^{AP \cup (T \times X)} \times 2^{AP} \cup T \times [k+1]$, where a letter in $2^{AP \cup (T \times X)}$, representing the original label, follows every sequence. When applying the construction in the proof of Theorem 6, we may ignore the letters in $2^{AP \cup (T \times X)}$ for the purposes of the intersection with the abstract system, but consider them for the transition relation.

Example 3. Let $s = \{a.x_1, b.x_1, c.x_2, d\}$ be a state in an abstract system, or a labeling of a transition in the Vardi-Wolper construction for some LTL formula over a set of atomic propositions. Then for the order $a < b < c$, for the case of the abstract system the translation of s to a GRA leads to the sequence of transitions $\{d\}\langle a, 1\rangle\langle b, 1\rangle\langle c, 2\rangle$. In the case of a VLTL formula, this sequence is followed by the letter $\{a.x_1, b.x_1, c.x_2\}$.

We now turn to use the translation of VLTL and abstract systems to GRAs in order identify a new fragment of VLTL for which model checking is decidable. For this, we turn to study a type of GRAs that is relevant for the translation. We define this type of GRA, and show that for GRAs that are a translation of VLTL formulas, nonemptiness is decidable. Further, we prove that this type is also decidable when considering an unary alphabet.

Consider a GRA $\mathcal{A} = \langle \Sigma \times D, \#, \langle r_1, r_2, r_3, \ldots r_k \rangle, \langle \#, \#, \ldots \# \rangle, Q, q_0, \delta, F \rangle$ with the following attributes:

- The guard condition is always the full inequality set.

[6] We present the alphabet this way to emphasize that the value attached to the letter in 2^{AP} does not matter.

- From q_0 exits a sequence of states S that assign values to $r_3, r_4, \ldots r_k$ with $\epsilon_\exists$ transitions, followed by a state s_k from which there is an $\epsilon_\forall$ transition that splits over all allowed values in r_1.
- From s_k, the GRA $\mathcal{A}$ behaves as an NRA without returning to the states of S, s_k or q_0.
- The content of the registers r_1 and $r_3 \ldots r_k$ does not change after the initial assignment,
- Every state of the NRA component has a self loop labeled $\langle \epsilon_\exists, 2 \rangle$. So in each step of a run, r_2 may change its content.

We denote GRAs of this type *single-split GRAs*. Recall that both the translation of a deterministic data automaton to a GRA and the GRA of Example 2 yield a single-split GRA with two registers, that is, the $\epsilon_\forall$ transition is taken from the intial state, and from there on the different copies continue their runs as runs of NRA. Moreover, notice that for VLTL formulas of the type $\exists x_1; \exists x_2; \ldots \exists x_k; \forall x; \varphi$ with a full inequality set, their translation to GRA also yields a single-split GRA with $k + 2$ registers.

Theorem 4. *The nonemptiness problem for single-split GRAs is undecidable.*

Nevertheless, there are sub-types of single-split GRAs for which nonemptiness is decidable. The first type are two-register single-split GRAs over an unary alphabet, that is, when $|\Sigma| = 1$.

A second type are *reverse-deterministic single-split GRAs*. An automaton is reverse-deterministic if by reversing its transitions we get a determinisic automaton. A single-split GRA is reverse-deterministic if its NRA component is reverse-deterministic with respect to the labeling of its edges (neglecting $\langle \epsilon_\exists, 2 \rangle$-transitions, that are in self-loops).

A third type of GRAs whose nonemptiness is decidable are GRAs for the intersection of a reverse-deterministic single-split GRA with an NRA.

For both reverse-deterministic single-split GRAs and for their intersection with NRAs, their decidable nonemptiness is essential for our purpose of studying decision problems for VLTL and abstract systems.

Theorem 5. *The nonemptiness problems for single-split GRAs over an unary alphabet, for reverse-deterministic single-split GRAs, and for the interesection of an NRA and a reverse-deterministic single-split GRA are decidable.*

Proof: *(sketch)* For all these types of GRA, we reduce their nonemptiness problem to the nonemptiness problem for multi-counter machines, for which nonemptiness is known to be decidable. Multi-counter machines comprise a state machine and a set of counters. Upon reading a letter, the machine advances on the state machine and increments or decrements some counter. The machine cannot perform zero-checks on the counter, and the run is stuck when it attempts to decrement a counter whose value is zero.

Given a GRA $\mathcal{A}$ of one of these types, the state machine of the multi-counter machine M simulates simultaneous runs on all copies of $\mathcal{A}$ after the $\epsilon_\forall$ split. Every state keeps the set of states of $\mathcal{A}$ in which the various copies are located, and each such state q is paired with a counter that keeps the number of copies that are currently in q. The runs

that do not handle a value that is included in the input word all behave similarly, and therefore it suffices to keep a single copy for all of them. Therefore, it suffices to track a finite number of copies. The counters are updated according to the transition relation of $\mathcal{A}$. The run accepts iff all nonempty counters are paired with accepting states.

The challenge in these constructions is to correctly update the counters. In general, since multi-counter machines do not allow zero checks on the counters, it is impossible to unite the content of two counters if the value they hold is unknown. Therefore, if in some transition of the GRA, two different states q and q' may move to the same state state s, updating the counters is impossible. However, we can show that for these three types of GRA, updating the counters is possible.

A single-split GRA over an unary alphabet can be reconstructed in such a way that in every step, the content of two counters is united only if the value of one of them is 1. In case of a reverse-deterministic GRA, no two counters have to be unified during the run.

For the intersection of a reverse-deterministic single-split GRA with an NRA, the challenge in the construction is to ensure that the progress of the single-split GRA agrees with the register assignment in the NRA. To achieve this, some of the states in the single-split GRA are paired with registers of the NRA. When a state q is paired with r_i, this means that one of the copies that is currently in q in the single-split GRA handles the value that is assigned to register r_i in the NRA. When the NRA reassigns a register r_i, a new state may be paired with r_i. Thus, the run progresses legally along both automata.

□

Consider a VLTL formula $\psi = \langle \alpha, E \rangle$ where $\alpha = \exists x_1; \exists x_2; \ldots \exists x_k; \forall x; \varphi$, and E is the full inequality set. Recall from Theorem 3 that ψ can be translated to a single-split GRA $\mathcal{A}_\psi$. Since the Vardi-Wolper construction for φ yields a reverse-deterministic automaton, we have that $\mathcal{A}_\psi$ is a reverse-deterministic single-split GRA. We can check the satisfiability of ψ for finite computations by checking the nonemptiness of $\mathcal{A}_\psi$. According to Theorem 5, this is decidable.

Similarly, recall from Theorem 3 that an abstract system with restricted computations can be translated to an NRA. Given a system and a VLTL formula, we can then decide the model checking problem by checking the nonemptiness of the intersection of the two matching GRAs – for the system, and for the negation of the formula. Consider a VLTL formula $\psi' = \langle \beta, E \rangle$ where $\beta = \forall x_1; \forall x_2; \ldots \forall x_k; \exists x; \varphi$ and E is the full inequality set. Then the negation of β is $\exists x_1; \exists x_2; \ldots \exists x_k; \forall x; \neg\varphi$, again yielding a reverse-deterministic single-split GRA $\mathcal{A}_{\neg\psi'}$ for $\neg\psi'$. According to Theorem 5, it is decidable whether the intersection of $\mathcal{A}_{\neg\psi'}$ with an NRA representing the system is nonempty, proving the decidability of the model-checking problem for this type of VLTL formulas. Therefore, we have the following.

Theorem 6. *Let $\psi = \langle \alpha, E \rangle$, where $\alpha = \exists x_1; \exists x_2; \ldots \exists x_k; \forall x; \varphi$, for a safety formula φ, and where E is the full inequality set.*

1. *It is decidable whether ψ is satisfiable.*
2. *Let S be an abstract system with restricted computations. It is decidable whether S satisfies $\neg\psi$.*

5 Discussion

GRAs offer an automata-theoretic approach to VLTL. By reasoning about GRAs, we can work towards closing the gap between the decidable and undecidable fragments and provide a full classification of the model-checking problem for VLTL. Indeed, the proof of undecidability of model checking of VLTL formulas with two $\exists$ quantifiers [7] can be altered to show the undecidability of satisfiability of VLTL formulas with two $\forall$ quantifiers, of satisfiability of VLTL formulas with a $\forall$ followed by an $\exists$ quantifier, and of model checking of VLTL formulas with an $\exists$ followed by a $\forall$ quantifier. Thus, the fragments considered in Theorem 6 complete the picture for the case of safety VLTL formulas. Here, we proved them to be decidable for the safe fragment of LTL. We are optimistic about an extension of our results here to the setting of infinite words and computations, which would lead to further decidable fragments. Finally, GRAs could also provide a framework for studying other formalisms that deal with infinite alphabets, such as XML and its related languages.

References

1. Mikolaj Bojanczyk and Slawomir Lasota. An extension of data automata that captures xpath. *Logical Methods in Computer Science*, 8(1), 2012.
2. Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS*, pages 7–16. IEEE Computer Society, 2006.
3. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
4. Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3), 2009.
5. D. Figueira. Alternating register automata on finite words and trees. *LMCS*, 8(1), 2012.
6. O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In *LATA*, pages 561–572. Springer, 2010.
7. Orna Grumberg, Orna Kupferman, and Sarai Sheinvald. Model checking systems and specifications with parameterized atomic propositions. In *ATVA*, pages 122–136, 2012.
8. M. Kaminski, and N. Francez. Finite-memory automata. *TCS*, 134(2):329–363, 1994.
9. M. and D. Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5):741–760, 2010.
10. Ahmet Kara, Thomas Schwentick, and Tony Tan. Feasible automata for two-variable logic with successor on data words. In *LATA*, pages 351–362, 2012.
11. A. Manuel and R. Ramanujam. Class counting automata on datawords. *Int. J. Found. Comput. Sci.*, 22(4):863–882, 2011.
12. F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In *MFCS '01*, pages 560–572, 2001.
13. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, 41–57, 2006.
14. T. Tan. *Pebble Automata for Data Languages: Separation, Decidability, and Undecidability*. PhD thesis, Technion - Computer Science Department, 2009.
15. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
16. Zhilin Wu. A decidable extension of data automata. In *GandALF*, pages 116–130, 2011.
17. Zhilin Wu. Commutative data automata. In *CSL*, pages 528–542, 2012.