Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems

Lukáš Holík¨radu, Radu Iosif, A Rogalewicz, T Vojnar

▶ **To cite this version:**

HAL Id: hal-02388322
https://hal.archives-ouvertes.fr/hal-02388322

Submitted on 1 Dec 2019

Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems

Lukáš Holík · **Radu Iosif** · **Adam Rogalewicz** ·
Tomáš Vojnar

Abstract A *generic register automaton* is a finite automaton equipped with variables (which may be viewed as counters or, more generally, registers) ranging over infinite data domains. A trace of a generic register automaton is an alternating sequence of alphabet symbols and values taken by the variables during an execution of the automaton. The problem addressed in this paper is the inclusion between the sets of traces (data languages) recognized by such automata. Since the problem is undecidable in general, we give a semi-algorithm based on a combination of abstraction refinement and antichains, which is proved to be sound and complete, but whose termination is not guaranteed. Moreover, we further enhance the proposed algorithm by exploiting a concept of data simulations, i.e., simulation relations aware of the data associated with the words. We have implemented our technique in a prototype tool and show promising results on multiple non-trivial examples.

L. Holík
FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
E-mail: holik@fit.vut.cz

R. Iosif
University Grenoble Alpes, CNRS, VERIMAG, Grenoble, France
E-mail: radu.iosif@univ-grenoble-alpes.fr

A. Rogalewicz
FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
Tel.: +420-54114-1188
E-mail: rogalew@fit.vut.cz

T. Vojnar
FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
E-mail: vojnar@fit.vut.cz

1 Introduction

Many results in formal languages and automata theory rely on the assumption that the alphabet over which languages are defined is finite. The finite alphabet hypothesis is crucial for the existence of determinization, complementation and decidability of language inclusion problems for the language acceptor class under consideration. However, this assumption prevents the use of automata as models of real-time systems or infinite-state programs. In general, traditional attempts to generalize classical finite-state automata to infinite alphabets, such as timed automata [2] or finite-memory register automata [24] face the complement closure problem: there exists automata for which the complement language cannot be recognized by automata in the same class. This prevents encoding language inclusion problems $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ as the emptiness of the language $\mathcal{L}(A) \cap \overline{\mathcal{L}(B)}$, because the complement $\overline{\mathcal{L}(B)}$ of the language $\mathcal{L}(B)$ cannot be computed within the class of A and B. Moreover, the language inclusion problem is proved to be undecidable, in general, for timed [2] and finite-memory [24] automata, unless severe restrictions are applied.

In this paper, we consider a generalization of finite-state automata, by adding finitely many variables that range over an infinite data domain and whose values are part of the language of the automaton. We address the *trace inclusion* problem between (i) a network of *generic register automata*[1] $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ that communicate via a set of input events $\Sigma_{\mathcal{A}}$ and a set of shared variables $\mathbf{x}_{\mathcal{A}}$, ranging over an infinite data domain, and (ii) a generic register automaton B whose set of variables $\mathbf{x}_B$ is a subset of $\mathbf{x}_{\mathcal{A}}$ and whose set of input events is $\Sigma_{\mathcal{B}}$. Here, by a *trace*, we understand an alternating sequence of valuations of the variables from the set $\mathbf{x}_B$ and input events from the set $\Sigma_{\mathcal{A}} \cap \Sigma_B$, starting and ending with a valuation. Typically, the automata network $\mathcal{A}$ models the implementation of a concurrent system and B is a specification of the set of good behaviors of the system. Then, a positive answer to the above inclusion problem means that the behavior of the implementation conforms to the specification, which is a natural verification problem.

Consider, for instance, the network $\langle A_1, \ldots, A_N \rangle$ of generic register automata equipped with the integer-valued variables x and v shown in Fig. 1–left. The automata synchronize on the **init** symbol and interleave their $\mathbf{a}_{1,\ldots,N}$ actions. Each automaton A_i increases the shared variable x and writes its identifier i into the shared variable v as long as the value of x is in the interval $[(i-1)\Delta, i\Delta - 1]$, and it is inactive outside this interval, where $\Delta \geqslant 1$ is an unbounded parameter of the network. A possible specification for this network might require that each firing sequence is of the form **init** $\mathbf{a}_{1,\ldots,N}^*$ $\mathbf{a}_2$ $\mathbf{a}_{2,\ldots,N}^* \ldots \mathbf{a}_i$ $\mathbf{a}_i^*$ for some $1 \leqslant i \leqslant N$, and that v is increased only on the first occurrence of the events $\mathbf{a}_2, \ldots, \mathbf{a}_i$, in this order. This condition is encoded by the automaton B (Fig. 1–right). Observe that only the v variable is shared between the network $\langle A_1, \ldots, A_N \rangle$ and the specification automaton B—we say that v is *observable* in this case. An example of a trace, for $\Delta = 2$ and $N \geqslant 3$, is: $(v = 0)$ **init** $(v = 1)$ $\mathbf{a}_1$ $(v = 1)$ $\mathbf{a}_1$ $(v = 1)$ $\mathbf{a}_2$ $(v = 2)$ $\mathbf{a}_2$ $(v = 2)$ $\mathbf{a}_3$ $(v = 3)$. Our problem is to check that this, and all other traces of the network, are included in the language of the specification automaton, called the *observer*. The trace inclusion problem has multiple applications, e.g.:

– Decision procedures for logics describing array structures within imperative programs [18, 17] that use a translation of array formulae to integer counter automata which encode the set of array models of a formula. The expressiveness of such logics is currently limited by the undecidability of the emptiness (reachability) problem for counter automata. If we give up on decidability, we can reduce an entailment between two array

---

[1] Generic register automata were called data automata in our preliminary work [22]. We have decided to change the name in order to avoid confusion with some other formalisms.
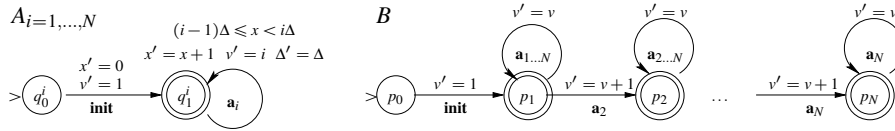
Fig. 1 An instance of the trace inclusion problem.

formulae to the trace inclusion of two integer counter automata, and use the method presented in this paper as a semi-decision procedure. To corroborate this claim, we have applied our trace inclusion method to several verification conditions for programs with unbounded arrays of integers [8].

– Timed automata and regular specifications of timed languages [2] can be both represented by finite automata extended with real-valued variables [15]. The verification problem boils down to the trace inclusion of two real-valued generic register automata. In this context, our method has been tested on several timed verification problems, including communication protocols and boolean circuits [32].

When developing a method for checking the inclusion between trace languages of automata extended with variables ranging over infinite data domains, the first problem is the lack of determinization and/or complementation results. In fact, certain classes of infinite state systems, such as finite-memory (register) [24] or timed automata [2], cannot be determinized and are provably not closed under complement. This is the case due to the fact that the values of the variables (registers, clocks) in such models of automata are not observable in the recognized language, that is determined by a series of internal computations.

However, if we allow the values of all variables of a generic register automaton to be part of its trace language, we obtain a determinization result, which generalizes the classical subset construction by taking into account the data valuations. Building on this first result, we define the complement of a trace language as an effectively computable generic register automaton. Thus, we can reduce the trace inclusion problem to the emptiness of a generic register product automaton $\mathcal{L}(A \times \overline{B}) = \varnothing$, just as in the finite alphabet case. However, the reduction of the trace inclusion to the emptiness problem crucially relies on the fact that the variables $\mathbf{x}_B$ of the right-hand side generic register automaton B (the one being determinized) are also controlled by the left-hand side automaton A, in other words, that B has no hidden variables. It is still an open problem whether and in which circumstances this reduction can be achieved in the presence of hidden variables.

The language emptiness problem for generic register automata is, in general, undecidable [28]. Nevertheless, several semi-algorithms and tools for this problem, better known as the *reachability* problem, have been developed [3, 20, 25, 16]. Among those, the technique of *lazy predicate abstraction* [20] combined with *counterexample-driven refinement* using *interpolants* [25] has been shown to be particularly successful in proving emptiness of infinite-state systems. Moreover, this technique shares similar aspects with *antichain-based algorithms for language inclusion* in the case of a finite alphabet [33, 1]. An important similarity is that both techniques use partial orders over the set of symbolic states, to prune the search space, by storing only incomparable such states. In other words, the successors of a covered state (in the sense of the partial order) are never explored, because any counterexample that could potentially arise from that state, can also be discovered by expanding the state that covers it.

Even if the trace inclusion problem can be reduced, under some conditions, to the emptiness of a counter automaton for which practical semi-algorithms exist, building the entire product counter automaton before checking its emptiness is usually not feasible. This is because the size of the product automaton is exponentially larger than the sum of the sizes of A and B, even when the trace alphabet is finite. Having, moreover, an infinite alphabet adds to the size of the product automaton, obtained by a generalization of the classical subset construction, used for determinization, in the finite alphabet case. Altogether, this prevents us from directly applying state-of-the-art methods and tool for checking emptiness of counter automata, or equivalently, nondeterministic integer programs, such as constrained Horn clause solvers [5].

To tackle this problem, we developed a semi-algorithm that builds the product automaton on-the-fly, while checking its emptiness. We achieve practical efficiency by combining the principle of antichain-based language inclusion algorithms [33,1] with the interpolant-based abstraction refinement semi-algorithm [25], via a general notion of language-based subsumption relation. This semi-algorithm has been first presented in our work [22]. Compared with that work, this paper includes more details and also proofs of the results.

Moreover, here we introduce a notion of *data simulations*, i.e., simulation relations on generic register automata, inspired by [27], and provide an algorithm to compute them. Further, we show how data simulations can be integrated into our trace inclusion semi-algorithm in order to improve its performance as done previously in the context of classical finite-alphabet automata [1].

We have implemented the trace inclusion semi-algorithm as well as its combination with data simulations in a prototype tool INCLUDER[2] and carried out a number of experiments, involving hardware, real-time systems, and array logic problems. The advantage of having a trace inclusion semi-algorithm is that we can write small automata-like specifications of the sets of good traces, instead of using, generally more complex, specifications of sets of erroneous behaviors.

1.1 An Overview of the Approach

We introduce the reader to our trace inclusion method by means of an example. Let us consider the network of generic register automata $\langle A_1, A_2 \rangle$ and the generic register automaton B from Fig. 1. We prove that, for any value of Δ, any trace of the network $\langle A_1, A_2 \rangle$, obtained as an interleaving of the actions of A_1 and A_2, is also a trace of the observer B. To this end, our procedure will fire increasingly longer sequences of input events, in search for a counterexample trace. We keep a set of predicates associated with each state $(\langle q_1, q_2 \rangle, P)$ of the product automaton where q_i is a state of A_i and P is a set of states of B. These predicates[3] are formulae that define over-approximations of the data values reached simultaneously by the network, when A_i is in the state q_i, and by the observer B, in every state from P.

The first input event is **init**, on which A_1 and A_2 synchronize, moving together from the initial state $\langle q_0^1, q_0^2 \rangle$ to $\langle q_1^1, q_1^2 \rangle$. In response, B can chose to either (i) move from $\{p_0\}$ to $\{p_1\}$, matching the only transition rule from p_0, or (ii) does not match the transition rule and move to the empty set. In the first case, the values of v match the relation of the rule $p_0 \xrightarrow{\text{init}, v'=1} p_1$, while in the second case, these values match the negated relation $\neg(v' = 1)$.

---

[2] http://www.fit.vutbr.cz/research/groups/verifit/tools/includer/

[3] Note that there is not a fixed set of predefined predicates. New predicates are discovered during refinement phase.
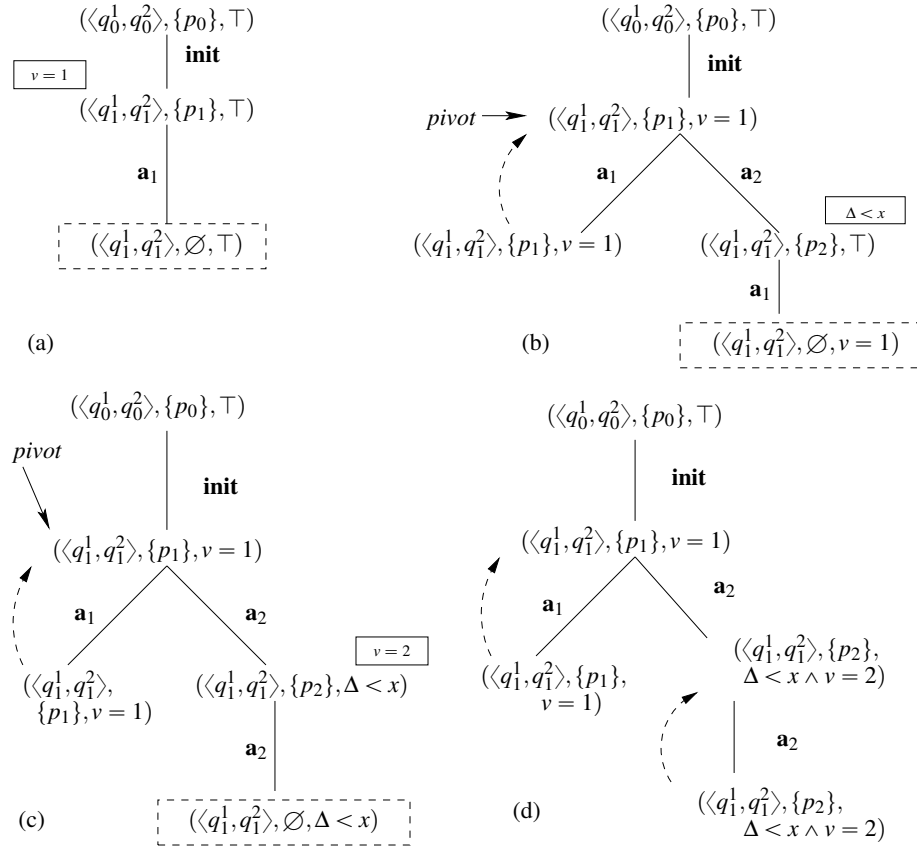
Fig. 2 A sample run of the proposed semi-algorithm.

The second case is impossible because the action of the network requires $x' = 0 \wedge v' = 1$. The only successor state is thus $(\langle q_1^1, q_1^2 \rangle, \{p_1\})$ in Fig. 2 (a). Since no predicates are initially available at this state, the best over-approximation of the set of reachable data valuations is the universal set, denoted as $\top$.

The second input event is $\mathbf{a}_1$, on which A_1 moves from q_1^1 back to itself, while A_2 makes an idle step because no transition with $\mathbf{a}_1$ is enabled from q_1^2. Again, B has the choice between moving from $\{p_1\}$ either to $\varnothing$ or $\{p_1\}$. Let us consider the first case, in which the successor state is $(\langle q_1^1, q_1^2 \rangle, \varnothing, \top)$. Since q_1^1 and q_1^2 are final states of A_1 and A_2, respectively, and no final state of B is present in $\varnothing$, we say that the state is accepting. If the accepting state (in dashed boxes in Fig. 2) is reachable according to the transition constraints along the input sequence **init.$\mathbf{a}_1$**, we have found a counterexample trace that is in the language of $\langle A_1, A_2 \rangle$ but not in the language of B.

To verify the reachability of the accepting state, we check the satisfiability of the path formula corresponding to the composition of the transition constraints $\theta_1 \equiv x' = 0 \wedge v' = 1$ (**init**) and $\theta_2 \equiv 0 \leqslant x < \Delta \wedge x' = x + 1 \wedge v' = 1 \wedge \neg(v' = v)$ ($\mathbf{a}_1$) in Fig. 2 (a). The formula $\theta_1 \wedge \theta_2$ is unsatisfiable, and the proof of infeasibility provides the interpolant $\langle v = 1 \rangle$. This formula is an explanation for the infeasibility of the path because it is implied by the con-

straint θ_1 and it is unsatisfiable in conjunction with the constraint θ_2. By associating the new predicate $v = 1$ with the state $(\langle q_1^1, q_1^2 \rangle, \{p_1\})$, we ensure that the same spurious path will never be explored again.

We delete the spurious counterexample and recompute the states along the input sequence **init**.$\mathbf{a}_1$ with the new predicate. In this case, $(\langle q_1^1, q_1^2 \rangle, \varnothing)$ is unreachable, and the outcome is $(\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1)$. However, this state was first encountered after the sequence **init**, so there is no need to store a second occurrence of this state in the tree. We say that the node **init**.$\mathbf{a}_1$ is subsumed by **init**, and indicate this by a dashed arrow in Fig. 2 (b).

We continue with $\mathbf{a}_2$ from the state $(\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1)$. In this case, A_1 makes an idle step and A_2 moves from q_1^2 to itself. In response, B has the choice between moving from $\{p_1\}$ to either (i) $\{p_1\}$ with the constraint $v' = v$, (ii) $\{p_2\}$ with the constraint $v' = v + 1$, (iii) $\{p_1, p_2\}$ with the constraint $v' = v \land v' = v + 1$ (this constraint is unsatisfiable, hence this case is discarded), (iv) $\varnothing$ for data values that satisfy $\neg(v' = v) \land \neg(v' = v + 1)$. The first and the last cases are also discarded because the value of v after **init** constrained to 1 and the A_2 imposes further the constraint $v' = 2$ and $v = 1 \land v' = 2 \land v' = v \to \bot$ for the first case and $v = 1 \land v' = 2 \land \neg(v' = v) \land \neg(v' = v + 1) \to \bot$. Hence, the only $\mathbf{a}_2$-successor of $(\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1)$ is $(\langle q_1^1, q_1^2 \rangle, \{p_2\}, \top)$, in Fig. 2 (b).

By firing the event $\mathbf{a}_1$ from this state, we reach $(\langle q_1^1, q_1^2 \rangle, \varnothing, v = 1)$, which is, again, accepting. We check whether the path **init**.$\mathbf{a}_2$.$\mathbf{a}_1$ is feasible, which turns out not to be the case. For efficiency reasons, we find the shortest suffix of this path that can be proved infeasible. It turns out that the sequence $\mathbf{a}_2$.$\mathbf{a}_1$ is infeasible starting from the state $(\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1)$, which is called the *pivot*. This proof of infeasibility yields the interpolant $\langle v = 1, \Delta < x \rangle$, and a new predicate $\Delta < x$ is associated with $(\langle q_1^1, q_1^2 \rangle, \{p_2\})$. The refinement phase rebuilds only the subtree rooted at the pivot state, in Fig. 2 (b).

The procedure then builds the tree in Fig. 2 (c) starting from the pivot node and finds the accepting state $(\langle q_1^1, q_1^2 \rangle, \varnothing, \Delta < x)$ as the result of firing the sequence **init**.$\mathbf{a}_2$.$\mathbf{a}_2$. This path is spurious, and the new predicate $v = 2$ is associated with the location $(\langle q_1^1, q_1^2 \rangle, \{p_2\})$. The pivot node is the same as in Fig. 2 (b), and, by recomputing the subtree rooted at this node with the new predicates, we obtain the tree in Fig. 2 (d), in which all frontier nodes are subsumed by their predecessors. Thus, no new event needs to be fired, and the procedure can stop reporting that the trace inclusion holds.

1.2 Related Work

Extending automata to deal with infinite alphabets is the purpose of the seminal work of Kaminski and Francez [24], who introduce *finite-memory automata* that accept languages over infinite alphabets using a finite set of registers that can be overwritten and compared for equality with the input. In addition, our model of generic register automata is parametric in the theory of the data used and allows comparisons between adjacent elements in the input stream. For instance, generic register automata can easily specify increasing sequences of integers, which is out of the scope of finite-memory automata. Moreover, the language inclusion problem is undecidable for finite-memory automata, if the right-hand side has more than 2 registers, while decidability is proved for at most 2 registers.

The trace inclusion problem has also been addressed in the context of timed automata [30]. Although the problem $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ is undecidable in general [2], decidability is recovered when the B automaton has at most one clock, or the only constant appearing in the clock constraints is zero. These are essentially the only known decidable cases of language inclusion for timed automata.

The study of *data automata* [6,13] usually deals with the complexity of decision problems in logics describing data languages for simple theories, typically infinite data domains with equality. Here, we focus on undecidable language inclusion problems between data automata controlled by generic first-order theories, by providing a semi-algorithm that proves to be effective, in practice.

Data words have also been considered in the context of *symbolic visibly pushdown automata* (SVPA) [12]. Language inclusion is decidable for SVPAs with transition guards from a decidable theory because SVPAs are closed under complement and the emptiness can be reduced to a finite number of queries expressible in the underlying theory of guards. Decidability comes here at the cost of reducing the expressivity and forbidding comparisons between adjacent positions in the input — here only comparisons between matching call/return positions of the input nested words are allowed.

Although trace inclusion cannot be reduced to the emptiness problem of automata from the same class in linear time, due to the exponential blowup caused by determinization, this is possible if one considers an alternating automaton model, such as the one introduced in [23]. This work generalizes from the trace inclusion problems considered in this paper, by considering unrestricted alternation. As an advantage, one can complement in linear time without the need for determinization. On the negative side, however, the emptiness check for alternating automata with variables is heavier than in our case because it relies on the ability of the SMT solver to answer queries in a combined theory of data and booleans. Due to this reason, on some test cases, the semi-algorithm [23] performs slower than the trace inclusion semi-algorithm presented here.

Several works on model checking infinite-state systems against CTL [4] and CTL* [10] specifications are related to our problem as they check inclusion between the set of computation trees of an infinite-state system and the set of trees defined by a branching temporal logic specification. First, the verification of existential CTL formulae [4] is reduced to solving forall-exists quantified Horn clauses by applying counterexample guided refinement to discover witnesses for existentially quantified variables. It is however not clear whether and how Horn clause solvers could be used for trace inclusion, which is a typical linear-time property, that requires an unbounded number of computation branches to synchronize on the same input word. To a very limited extent, for alphabets consisting of one symbol, one can encode the (non-)emptiness of alternating automata as the existence of solutions of a system of Horn clauses [?, Section 7.2.3]. However this encoding fails for alphabets of size two or more, let alone for infinite data alphabets. Moreover, we have not encountered a polynomial-time encoding of trace inclusion as a system of Horn clauses in the existing literature[4].

Finally, the work [10] on CTL* verification of infinite systems is based on partial symbolic determinization, using prophecy variables to summarize the future program execution. For finite alphabets, automata are a strictly more expressive formalism than temporal logics[5]. Such a comparison is, however, non-trivial for languages over infinite alphabets. However, in practice, we found the generic register automata considered in this paper to be a natural tool for specifying verification conditions of array programs [18,17,8] and regular properties of timed languages [2].

---

[4] Our reduction to the emptiness of product automata is at least exponential.

[5] For (in)finite words, the class of LTL-definable languages coincides with the star-free languages, which are a strict subclass of (ω-)regular languages.

1.3 Organization of the Paper

The rest of the paper is organized as follows: Section 2 describes preliminaries. Section 3 discusses closure properties of the considered class of generic register automata. Section 4 describes the trace inclusion semi-algorithm. Section 5 presents a concept of simulation relations on register automata and their integration into the proposed trace inclusion algorithm. Section 6 is an overview of our experimental evaluation, and, finally, Section 7 concludes the paper.

2 Preliminary Definitions

Let $\mathbb{N}$ denote the set of non-negative integers including zero. For any $k, \ell \in \mathbb{N}$, $k \leqslant \ell$, we write $[k, \ell]$ for the set $\{k, k+1, \ldots, \ell\}$. We write $\bot$ and $\top$ for the boolean constants *false* and *true*, respectively. Given a possibly infinite data domain $\mathcal{D}$, we denote by $\mathrm{Form}(\mathcal{D}) = \langle \mathcal{D}, f_1, \ldots, f_m \rangle$ the set of syntactically correct first-order formulae with function symbols $f_1, \ldots, f_m$. A variable x is said to be *free* in a formula ϕ, denoted as $\phi(x)$, iff it does not occur under the scope of a quantifier.

Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be a finite set of variables. A *valuation* $\nu : \mathbf{x} \to \mathcal{D}$ is an assignment of the variables in $\mathbf{x}$ with values from $\mathcal{D}$. We denote by $\mathcal{D}^{\mathbf{x}}$ the set of such valuations. For a formula $\phi(\mathbf{x})$, we denote by $\nu \models \phi$ the fact that substituting in ϕ each variable $x \in \mathbf{x}$ by $\nu(x)$ yields a valid formula in the first-order theory of $\mathrm{Form}(\mathcal{D})$. In this case, ν is said to be a *model* of ϕ. A formula is said to be *satisfiable* iff it has a model. For a formula $\phi(\mathbf{x}, \mathbf{x}')$ where $\mathbf{x}' = \{x' \mid x \in \mathbf{x}\}$ and two valuations $\nu, \nu' \in \mathcal{D}^{\mathbf{x}}$, we denote by $(\nu, \nu') \models \phi$ the fact that the formula obtained from ϕ by substituting each x with $\nu(x)$ and each x' with $\nu'(x')$ is valid in the first-order theory of $\mathrm{Form}(\mathcal{D})$.

2.1 Generic Register Automata

Generic register automata[6] (GRA) are extensions of non-deterministic finite automata with variables ranging over an infinite data domain $\mathcal{D}$ with the first-order theory of $\mathrm{Form}(\mathcal{D})$. Formally, a GRA is a tuple $A = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q, \iota, F, \Delta \rangle$, where:
 - Σ is a finite alphabet of input events and $\diamond \in \Sigma$ is a special padding symbol,
 - $\mathbf{x} = \{x_1, \ldots, x_n\}$ is a set of variables,
 - Q is a finite set of *states*, $\iota \in Q$ is an *initial* state, $F \subseteq Q$ are *final* states, and
 - Δ is a set of *rules* of the form $q \xrightarrow{\sigma, \phi(\mathbf{x}, \mathbf{x}')} q'$ where $\sigma \in \Sigma$ is an alphabet symbol and $\phi(\mathbf{x}, \mathbf{x}')$ is a formula in $\mathrm{Form}(\mathcal{D})$.

A *configuration* of A is a pair $(q, \nu) \in Q \times \mathcal{D}^{\mathbf{x}}$. We say that a configuration (q', ν') is a *successor* of (q, ν) if and only if there exists a rule $q \xrightarrow{\sigma, \phi} q' \in \Delta$ and $(\nu, \nu') \models \phi$. We denote the successor relation by $(q, \nu) \xrightarrow{\sigma, \phi}_A (q', \nu')$, and we omit writing ϕ and A when no confusion may arise. We denote by $succ_A(q, \nu) = \{(q', \nu') \mid \exists \sigma \in \Sigma : (q, \nu) \xrightarrow{\sigma}_A (q', \nu')\}$ the set of successors of a configuration (q, ν).

For any $n \geqslant 0$, a *trace* is a finite sequence $w = (\nu_0, \sigma_0), \ldots, (\nu_{n-1}, \sigma_{n-1}), (\nu_n, \diamond)$ of pairs (ν_i, σ_i) taken from the infinite alphabet $\mathcal{D}^{\mathbf{x}} \times \Sigma$ [if $n = 0$, the trace is just $(\nu_0, \diamond)$]. A *run* of A over the *trace* w is a sequence of configurations $\pi : (q_0, \nu_0) \xrightarrow{\sigma_0} (q_1, \nu_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-1}} (q_n, \nu_n)$ [for $n = 0$, the run is (q_0, ν_0) only]. We say that the run π is *accepting* if and only if $q_n \in F$,

---

 [6] Called data automata in [22].

in which case A *accepts* w. The *language* of A, denoted $\mathcal{L}(A)$, is the set of traces accepted by A.

2.2 Generic Register Automata Networks

A *generic register automata network* (GRAN) is a non-empty tuple $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ of generic register automata $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$, $i \in [1, N]$ whose sets of states are pairwise disjoint. A GRAN is a succint representation of an exponentially larger GRA $\mathcal{A}^e = \langle \mathcal{D}, \Sigma_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}}, Q_{\mathcal{A}}, \iota_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}} \rangle$, called the *expansion* of $\mathcal{A}$, where:

- $\Sigma_{\mathcal{A}} = \Sigma_1 \cup \ldots \cup \Sigma_N$ and $\mathbf{x}_{\mathcal{A}} = \mathbf{x}_1 \cup \ldots \cup \mathbf{x}_N$,
- $Q_{\mathcal{A}} = Q_1 \times \ldots \times Q_N$, $\iota_{\mathcal{A}} = \langle \iota_1, \ldots, \iota_N \rangle$ and $F_{\mathcal{A}} = F_1 \times \ldots \times F_N$,
- $\langle q_1, \ldots, q_N \rangle \xrightarrow{\sigma, \varphi} \langle q'_1, \ldots, q'_N \rangle$ if and only if there exists a set of indices $I \subseteq [1, N]$ such that (i) for all $i \in I$, $q_i \xrightarrow{\sigma, \varphi_i} q'_i$, (ii) for all $i \notin I$, $q_i = q'_i$, and (iii) $\varphi \equiv \bigwedge_{i \in I} \varphi_i \wedge \bigwedge_{j \notin I} \tau_j$, where $I = \{i \in [1, N] \mid q_i \xrightarrow{\sigma, \varphi_i} q'_i \in \Delta_i\}$ is the set of GRA that can move from q_i to q'_i while reading the input symbol σ, and $\tau_j \equiv \bigwedge_{x \in \mathbf{x}_j \setminus (\bigcup_{i \in I} \mathbf{x}_i)} x' = x$ propagates the values of the local variables in A_j that are not updated by $\{A_i\}_{i \in I}$.

Intuitively, all automata that can read an input symbol synchronize their actions on that symbol whereas the rest of the automata make an idle step and copy the values of their local variables which are not updated by the active automata. The language of the GRAN $\mathcal{A}$ is defined as the language of its expansion GRA, i.e., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^e)$.

2.3 Trace Inclusion

Let $\mathcal{A}$ be a GRAN and $\mathcal{A}^e = \langle \mathcal{D}, \Sigma, \mathbf{x}_{\mathcal{A}}, Q_{\mathcal{A}}, \iota_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}} \rangle$ be its expansion. For a set of variables $\mathbf{y} \subseteq \mathbf{x}_{\mathcal{A}}$, we denote by $\nu \!\downarrow_{\mathbf{y}}$ the restriction of a valuation $\nu \in \mathcal{D}^{\mathbf{x}_{\mathcal{A}}}$ to the variables in $\mathbf{y}$. For a trace $w = (\nu_0, \sigma_0), \ldots, (\nu_n, \diamond) \in (\mathcal{D}^{\mathbf{x}_{\mathcal{A}}} \times \Sigma_{\mathcal{A}})^*$, we denote by $w \!\downarrow_{\mathbf{y}}$ the trace $(\nu_0 \!\downarrow_{\mathbf{y}}, \sigma_0), \ldots,$ $(\nu_{n-1} \!\downarrow_{\mathbf{y}}, \sigma_{n-1}), (\nu_n \!\downarrow_{\mathbf{y}}, \diamond) \in (\mathcal{D}^{\mathbf{y}} \times \Sigma)^*$. We lift this notion to sets of words in the natural way, by defining $\mathcal{L}(\mathcal{A}) \!\downarrow_{\mathbf{y}} = \{w \!\downarrow_{\mathbf{y}} \mid w \in \mathcal{L}(\mathcal{A})\}$.

We are now ready to define the trace inclusion problem on which we focus in this paper. Given a GRAN $\mathcal{A}$ as before and a GRA $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ such that $\mathbf{x}_B \subseteq \mathbf{x}_{\mathcal{A}}$, the *trace inclusion problem* asks whether $\mathcal{L}(\mathcal{A}) \!\downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$? The right-hand side GRA B is called *observer*, and the variables in $\mathbf{x}_B$ are called *observable* variables.

3 Boolean Closure Properties of Generic Register Automata

We show first that generic register automata are closed under the boolean operations of union, intersection and complement and that they are amenable to determinization. Clearly, the emptiness problem is, in general, undecidable, due to the result of Minsky on 2-counter machines with integer variables, increment, decrement and zero test [28].

Let $A = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q, \iota, F, \Delta \rangle$ be a GRA for the rest of this section. A is said to be *deterministic* if and only if, for each trace $w \in \mathcal{L}(A)$, A has at most one run over w. The first result of this section is that, interestingly, any GRA can be determinized while preserving its language. The determinization procedure is a generalization of the classical subset construction for Rabin-Scott word automata on finite alphabets. The reason why determinization is possible for automata over an infinite data alphabet $\mathcal{D}^{\mathbf{x}} \times \Sigma$ is that the successive values taken by *each variable* $x \in \mathbf{x}$ are tracked by the language $\mathcal{L}(A) \subseteq (\mathcal{D}^{\mathbf{x}} \times \Sigma)^*$. This assumption is

crucial: a typical example of automata over an infinite alphabet, that cannot be determinized, are timed automata [2], where only the elapsed time is reflected in the language, and not the values of the variables (clocks).

Formally, the *deterministic* GRA accepting the language $\mathcal{L}(A)$ is defined as $A^d = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q^d, \iota^d, F^d, \Delta^d \rangle$, where $Q^d = 2^Q$, $\iota^d = \{\iota\}$, $F^d = \{P \subseteq Q \mid P \cap F \neq \varnothing\}$ and Δ^d is the set of rules $P \xrightarrow{\sigma, \theta} P'$ such that the formula:

$$\theta(\mathbf{x}, \mathbf{x}') = \bigwedge_{p' \in P'} \bigvee_{\substack{p \xrightarrow{\sigma, \psi} p' \in \Delta \\ p \in P}} \psi \wedge \bigwedge_{p' \in Q \setminus P'} \bigwedge_{\substack{p \xrightarrow{\sigma, \phi} p' \in \Delta \\ p \in P}} \neg \phi$$

is satisfiable[7]. The main difference with the classical subset construction for Rabin-Scott automata is that here we consider *all sets* P' of states that have a predecessor in P, not just the maximal such set. This refined subset construction takes into account not just the alphabet symbols in Σ, but also the valuations of the variables in $\mathbf{x}$. Observe, moreover, that A^d can be built for any first-order theory of Form($\mathcal{D}$) that is closed under conjunction, disjunction, and negation. The following lemma states the main properties of A^d.

Lemma 1 *Given a GRA $A = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q, \iota, F, \Delta \rangle$, (1) for any $w \in (\mathcal{D}^{\mathbf{x}} \times \Sigma)^*$ and $P \in Q^d$, A^d has exactly one run on w that starts in P, and (2) $\mathcal{L}(A) = \mathcal{L}(A^d)$.*

Proof (1) Let $w = (\nu_0, \sigma_0), \ldots, (\nu_{n-1}, \sigma_{n-1}), (\nu_n, \diamond)$ be an arbitrary trace and $P \subseteq Q$ be a state of A^d. We first build a run $\pi = (P_0, \nu_0) \xrightarrow{\sigma_0, \theta_0} (P_1, \nu_1) \ldots \xrightarrow{\sigma_{n-1}, \theta_{n-1}} (P_n, \nu_n)$ of A^d such that $P_0 = P$, by induction on $n \geqslant 0$. If $n = 0$, then $w = (\nu_0, \diamond)$ and $\pi = (P_0, \nu_0)$ is trivially a run of A^d over w. For the induction step, let $n > 0$ and suppose that A^d has a run $(P_0, \nu_0) \xrightarrow{\sigma_0, \theta_0} \ldots (P_{n-1}, \nu_{n-1})$ such that $P_0 = P$. We extend this run to a run over w by considering:

$$P_n = \left\{ p \in Q \mid \exists q \in P_{n-1} \,.\, q \xrightarrow{\sigma_{n-1}, \phi} p \in \Delta \text{ and } (\nu_{n-1}, \nu_n) \models \phi \right\},$$
$$\theta_n \equiv \bigwedge_{p' \in P_n} \bigvee_{\substack{p \xrightarrow{\sigma, \psi} p' \in \Delta \\ p \in P_{n-1}}} \psi \wedge \bigwedge_{p' \in Q \setminus P_n} \bigwedge_{\substack{p \xrightarrow{\sigma, \phi} p' \in \Delta \\ p \in P_{n-1}}} \neg \phi.$$

It is not hard to see that $(\nu_{n-1}, \nu_n) \models \theta_n$, thus $(P_0, \nu_0) \xrightarrow{\sigma_0, \theta_0} \ldots \xrightarrow{\sigma_n, \theta_n} (P_n, \nu_n)$ is indeed a run of A^d over w. To show that π is unique, suppose, by contradiction, that there exists a different run $\pi' = (R_0, \nu_0) \xrightarrow{\sigma_0, \omega_0} (R_1, \nu_1) \ldots \xrightarrow{\sigma_{n-1}, \omega_{n-1}} (R_n, \nu_n)$ such that $P_0 = R_0 = P$. Notice that the relation labeling any transition rule $P_i \xrightarrow{\sigma_i, \theta_i} P_{i+1}$ is entirely determined by the sets P_i and P_{i+1}, so two runs are different iff they differ in at least one state, i.e., $P_j \neq R_j$ for some $j \in [1, n]$. Let i denote the smallest such j and suppose that there exists $p \in P_i$ such that $p \notin R_i$ (the symmetrical case $p \in R_i$ and $p \notin P_i$ is left to the reader). By the definition of Δ^d, there exists $q \in P_{i-1} = R_{i-1}$ such that $q \xrightarrow{\sigma_{i-1}, \psi} p \in \Delta$. Since $(\nu_{i-1}, \nu_i) \models \theta_{i-1} \wedge \omega_{i-1}$, we obtain that $(\nu_{i-1}, \nu_i) \models \bigvee\{\psi \mid q \xrightarrow{\sigma_{i-1}, \psi} p \in \Delta, q \in P_{i-1}\}$ and $(\nu_{i-1}, \nu_i) \models \bigwedge\{\neg \psi \mid q \xrightarrow{\sigma_{i-1}, \psi} p \in \Delta, q \in P_{i-1}\}$, contradiction. Thus π is the only run of A^d over w starting in P.

(2) Let $w = (\nu_0, \sigma_0), \ldots, (\nu_{n-1}, \sigma_{n-1}), (\nu_n, \diamond)$ be a trace. "$\subseteq$" If $w \in \mathcal{L}(A)$, then A has a run $(q_0, \nu_0) \xrightarrow{\sigma_0, \phi_0} \ldots \xrightarrow{\sigma_{n-1}, \theta_{n-1}} (q_n, \nu_n)$ such that $q_0 = \iota$ and $q_n \in F$. By Point 1, A^d has a unique run $(P_0, \nu_0) \xrightarrow{\sigma_0, \theta_0} \ldots \xrightarrow{\sigma_{n-1}, \theta_{n-1}} (P_n, \nu_n)$ over w. We prove that $q_i \in P_i$ by induction on $i \in [0, n]$. For $i = 0$, we have $P_0 = \{\iota\}$ by the definition of A^d. For the induction step, suppose that $i \in [1, n]$ and $q_{i-1} \in P_{i-1}$. By contradiction, assume that $q_i \notin P_i$. Since $(\nu_{i-1}, \nu_i) \models \theta_{i-1}$,

---

[7] Note that the empty disjunction is equivalent to $\bot$. Hence $\theta(\mathbf{x}, \mathbf{x}')$ satisfiable implies that for all $p' \in P'$ there exists $p \in P$ and a rule $p \xrightarrow{\sigma, \psi} p' \in \Delta$.

we obtain $(\nu_{i-1}, \nu_i) \models \neg\phi_{i-1}$, contradiction. Thus $q_i \in P_i$ for all $i \in [0, n]$, and $q_n \in P_n$, hence $P_n \cap F \neq \varnothing$. Then $P_n \in F^d$, and $w \in \mathcal{L}(A^d)$. "$\supseteq$" If $w \in \mathcal{L}(A^d)$, then A^d has a (unique) run $(P_0, \nu_0) \xrightarrow{\sigma_0, \theta_0} (P_1, \nu_1) \ldots \xrightarrow{\sigma_{n-1}, \theta_{n-1}} (P_n, \nu_n)$ over w such that $P_0 = \{\iota\}$ and $P_n \cap F \neq \varnothing$. Then there exists $p_n \in P_n \cap F$, and, by the definition of A^d, there exists $p_{n-1} \in P_{n-1}$ such that $p_{n-1} \xrightarrow{\sigma_{n-1}, \psi_{n-1}} p_n \in \Delta$ and $(\nu_{n-1}, \nu_n) \models \psi_{n-1}$. Continuing this argument backwards, we can find a run $(q_0, \nu_0) \xrightarrow{\sigma_0, \psi_0} \ldots \xrightarrow{\sigma_{n-1}, \psi_{n-1}} (q_n, \nu_n)$ of A over w such that $q_i \in P_i$ for all $i \in [0, n]$. Since $P_0 = \{\iota\}$ and $q_n \in F$, we obtain that $w \in \mathcal{L}(A)$. $\qquad\square$

The construction of a deterministic GRA recognizing the language of A is key to defining a GRA that recognizes the complement of A. Let $\overline{A} = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q^d, \iota^d, Q^d \backslash F^d, \Delta^d \rangle$. In other words, $\overline{A}$ has the same structure as A^d, and the set of final states consists of those subsets that contain no final state, i.e., $\{P \subseteq Q \mid P \cap F = \varnothing\}$. Using Lemma 1, it is not difficult to show that $\mathcal{L}(\overline{A}) = (\mathcal{D}^{\mathbf{x}} \times \Sigma)^* \backslash \mathcal{L}(A)$.

Next, we show closure of GRA under intersection. Let $B = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q', \iota', F', \Delta' \rangle$ be a GRA and define $A \times B = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q \times Q', (\iota, \iota'), F \times F', \Delta^{\times} \rangle$ where $(q, q') \xrightarrow{\sigma, \varphi} (p, p') \in \Delta^{\times}$ if and only if $q \xrightarrow{\sigma, \phi} p \in \Delta$, $q' \xrightarrow{\sigma, \psi} p' \in \Delta'$ and $\varphi \equiv \phi \wedge \psi$. It is easy to show that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$. GRA are also closed under union since $\mathcal{L}(A) \cup \mathcal{L}(B) = \mathcal{L}(\overline{\overline{A} \times \overline{B}})$.

Let us turn now to the trace inclusion problem. The following lemma shows that this problem can be effectively reduced to an equivalent language emptiness problem. However, note that this reduction does not work when the trace inclusion problem is generalized by removing the condition $\mathbf{x}_B \subseteq \mathbf{x}_A$. In other words, if the observer uses local variables not shared with the network[8], i.e., $\mathbf{x}_B \backslash \mathbf{x}_A \neq \varnothing$, the generalized trace inclusion problem $\mathcal{L}(A){\downarrow}_{\mathbf{x}_A \cap \mathbf{x}_B} \subseteq \mathcal{L}(B){\downarrow}_{\mathbf{x}_A \cap \mathbf{x}_B}$ has a negative answer iff *there exists a trace* $w = (\nu_0, \sigma_0), \ldots, (\nu_n, \diamond) \in \mathcal{L}(A)$ such that, *for all valuations* $\mu_0, \ldots, \mu_n \in \mathcal{D}^{\mathbf{x}_B \backslash \mathbf{x}_A}$, we have $w' = (\nu_0 {\downarrow}_{\mathbf{x}_A \cap \mathbf{x}_B} \cup \mu_0, \sigma_0), \ldots, (\nu_n {\downarrow}_{\mathbf{x}_A \cap \mathbf{x}_B} \cup \mu_n, \diamond) \notin \mathcal{L}(B)$. This kind of quantifier alternation cannot be easily accommodated within the framework of language emptiness, in which only one type of (existential) quantifier occurs.

Lemma 2 *Given GRA* $A = \langle \mathcal{D}, \Sigma, \mathbf{x}_A, Q_A, \iota_A, F_A, \Delta_A \rangle$ *and* $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ *such that* $\mathbf{x}_B \subseteq \mathbf{x}_A$. *Then* $\mathcal{L}(A){\downarrow}_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$ *if and only if* $\mathcal{L}(A \times \overline{B}) = \varnothing$.

Proof We have $\mathcal{L}(A){\downarrow}_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$ iff $\mathcal{L}(A) {\downarrow}_{\mathbf{x}_B} \cap \mathcal{L}(\overline{B}) = \mathcal{L}(A \times \overline{B}){\downarrow}_{\mathbf{x}_B} = \varnothing$ iff $\mathcal{L}(A \times \overline{B}) = \varnothing$. $\qquad\square$

The trace inclusion problem is undecidable, which can be shown by reduction from the language emptiness problem for GRA (take B such that $\mathcal{L}(B) = \varnothing$). However, the above lemma shows that any semi-decision procedure for the language emptiness problem can also be used to deal with the trace inclusion problem.

4 Abstract, Check, and Refine for Trace Inclusion

This section describes our semi-algorithm for checking the trace inclusion between a given network $\mathcal{A}$ and an observer B. Let $\mathcal{A}^e$ denote the expansion of $\mathcal{A}$, defined in Section 2. In the light of Lemma 2, the trace inclusion problem $\mathcal{L}(\mathcal{A}){\downarrow}_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$, where the set of observable

---

[8] For timed automata, this is the case since the only shared variable is the time, and the observer may have local clocks.

variables $\mathbf{x}_B$ is included in the set of network variables, can be reduced to the language emptiness problem $\mathcal{L}(\mathcal{A}^e \times \overline{B}) = \varnothing$.

Although language emptiness is in general undecidable for generic register automata [28], several cost-effective semi-algorithms and tools [19,25,16,3] have been developed, showing that it is possible, in many practical cases, to provide a yes/no answer to this problem. However, to apply one of the existing off-the-shelf tools to our problem, one needs to build the product automaton $\mathcal{A}^e \times \overline{B}$ prior to the analysis. Due to the inherent state explosion caused by the interleaving semantics of the network as well as by the complementation of the observer, such a solution would not be efficient in practice.

To avoid building the product automaton, our procedure builds *on-the-fly* an over-approximation of the (possibly infinite) set of reachable configurations of $\mathcal{A}^e \times \overline{B}$. This over-approximation is defined using the approach of *lazy predicate abstraction* [19], combined with *counterexample-driven abstraction refinement* using *interpolants* [25]. We store the explored abstract states in a structure called an *antichain tree*. In general, antichain-based algorithms [33,1] store only states which are incomparable wrt a partial order called *subsumption*. Our method can be thus seen as an extension of the antichain-based language inclusion algorithms [33,1] to infinite state systems by means of predicate abstraction and interpolation-based refinement. Since the trace inclusion problem is undecidable in general, termination of our procedure is not guaranteed; in the following, we shall, however, call our procedure an algorithm for the sake of brevity.

4.1 Antichain Trees

In this section, we define antichain trees, which are the main data structure of the trace inclusion (semi-)algorithm. Let $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ be a network of automata where $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$, for all $i \in [1, N]$, and let $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ be an observer such that $\mathbf{x}_B \subseteq \bigcup_{i=1}^N \mathbf{x}_i$. We also denote by $\mathcal{A}^e = \langle \mathcal{D}, \Sigma_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}}, Q_{\mathcal{A}}, \iota_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}} \rangle$ the expansion of the network $\mathcal{A}$ and by $\mathcal{A}^e \times \overline{B} = \langle \mathcal{D}, \Sigma_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}}, Q^p, \iota^p, F^p, \Delta^p \rangle$ the product automaton used for checking language inclusion.

An *antichain tree* for the network $\mathcal{A}$ and the observer B is a tree whose nodes are labeled by *product states* (see Fig. 2 for examples)[9]. Intuitively, a product state is an over-approximation of the set of reachable configurations of the product automaton $\mathcal{A}^e \times \overline{B}$ that share the same control state. Formally, a *product state for $\mathcal{A}$ and B* is defined as a tuple $s = (\mathbf{q}, P, \Phi)$ where (i) $(\mathbf{q}, P)$ is a state of $\mathcal{A}^e \times \overline{B}$ with $\mathbf{q} = \langle q_1, \ldots, q_N \rangle$ being a state of the network expansion $\mathcal{A}^e$ and P being a set of states of the observer B, and (ii) $\Phi(\mathbf{x}_{\mathcal{A}}) \in \text{Form}(\mathcal{D})$ is a formula which defines an over-approximation of the set of valuations of the variables $\mathbf{x}_{\mathcal{A}}$ that reach the state $(\mathbf{q}, P)$ in $\mathcal{A}^e \times \overline{B}$. A product state $s = (\mathbf{q}, P, \Phi)$ is a finite representation of a possibly infinite set of configurations of $\mathcal{A}^e \times \overline{B}$, denoted as $[[s]] = \{(\mathbf{q}, P, \nu) \mid \nu \models \Phi\}$. [10]

To build an over-approximation of the set of reachable states of the product automaton, we need to compute, for a product state s, an over-approximation of the set of configurations

---

[9] The formal definition of antichain trees will be given as Definition 1 later in this section.

[10] Note that the above choice of the product state in the form $s = (\mathbf{q}, P, \Phi)$ is not straightforward and resulted from several previous unsuccessful attempts. For example, if one chooses to associate separate formulae for the valuations of the variables with $\mathbf{q}$ and each of the states in P, which seems to be a quite natural choice, the construction becomes unsound. Intuitively, when a successor state of such a product state is computed, the disjunction of the formulae joint with the successors of P may entail the formula joint with the successor of $\mathbf{q}$. However, that does not mean that all pairs of source/target valuations possible in $\mathcal{A}^e$ are possible in $\overline{B}$ too. More details are provided in Appendix A.

that can be reached in one step from s. To this end, we define first a finite abstract domain of product states, based on the notion of *predicate map*. A predicate map is a partial function that associates sets of facts[11] about the values of the variables used in the product automaton with components of a product state. Facts are called *predicates* and components of a product state are called *substates*. Formally, a *substate* of a state $(\langle q_1, \ldots, q_N \rangle, P) \in Q^p$ of the product automaton $\mathcal{A}^e \times \overline{B}$ is a pair $(\langle q_{i_1}, \ldots, q_{i_k} \rangle, S)$ such that (i) $\langle q_{i_1}, \ldots, q_{i_k} \rangle$ is a subsequence of $\langle q_1, \ldots, q_N \rangle$, and (ii) $S \neq \varnothing$ only if $S \cap P \neq \varnothing$.

The reason behind the distribution of predicates over substates is two-fold. First, we would like the abstraction to be *local*, i.e., the predicates needed to define a certain subtree in the antichain must be associated with the labels of that subtree only. Second, once a predicate appears in the context of a substate, it should be subsequently reused whenever that same substate occurs as part of another product state.

We denote the substate relation by $(\langle q_{i_1}, \ldots, q_{i_k} \rangle, S) \lhd (\langle q_1, \ldots, q_N \rangle, P)$. The substate relation requires the automata $A_{i_1}, \ldots, A_{i_k}$ of the network $\mathcal{A}$ to be in the control states $q_{i_1}, \ldots, q_{i_k}$ simultaneously, and the observer B to be in at least some state of S provided $S \neq \varnothing$ (if $S = \varnothing$, the state of B is considered to be irrelevant). Let $\mathcal{S}_{\langle \mathcal{A}, B \rangle} = \{ r \mid \exists q \in Q^p \ . \ r \lhd q \}$ be the set of substates of a state of $\mathcal{A}^e \times \overline{B}$.

A *predicate map* $\Pi : \mathcal{S}_{\langle \mathcal{A}, B \rangle} \to 2^{\text{Form}(\mathcal{D})}$ associates each substate $(\mathbf{r}, S) \in Q_{i_1} \times \ldots \times Q_{i_k} \times 2^{Q_B}$ with a set of formulae $\pi(\mathbf{x})$ where (i) $\mathbf{x} = \mathbf{x}_{i_1} \cup \ldots \cup \mathbf{x}_{i_k} \cup \mathbf{x}_B$ if $S \neq \varnothing$, and (ii) $\mathbf{x} = \mathbf{x}_{i_1} \cup \ldots \cup \mathbf{x}_{i_k}$ if $S = \varnothing$. Notice that a predicate associated with a substate refers only to the local variables of those network components $A_{i_1}, \ldots, A_{i_k}$ and of the observer B that occur in the particular substate.

Example 1 The antichain in Fig. 2 (d) uses the predicate map $(\langle q_1^1, q_1^2 \rangle, \{p_1\}) \mapsto \{v = 1\}$, $(\langle q_1^1, q_1^2 \rangle, \{p_2\}) \mapsto \{\Delta < x, v = 2\}$. ∎

We are now ready to define the abstract semantics of the product automaton $\mathcal{A}^e \times \overline{B}$, induced by a given predicate map. For convenience, we define first a set *Post*(s) of *concrete successors* of a product state $s = (\mathbf{q}, P, \Phi)$ such that $(\mathbf{r}, S, \Psi) \in Post(s)$ if and only if (i) the product automaton $\mathcal{A}^e \times \overline{B}$ has a rule $(\mathbf{q}, P) \xrightarrow{\sigma, \theta} (\mathbf{r}, S) \in \Delta^p$ and $\Psi(\mathbf{x}_{\mathcal{A}}) \equiv \exists \mathbf{x}'_{\mathcal{A}} \ . \ \Phi(\mathbf{x}'_{\mathcal{A}}) \wedge \theta(\mathbf{x}'_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}})$ is satisfiable. The set of concrete successors does not contain states with empty set of valuations because these states are unreachable in $\mathcal{A}^e \times \overline{B}$.

Given a predicate map Π, the set *Post*$_\Pi(s)$ of *abstract successors* of a product state s is defined as follows: $(\mathbf{r}, S, \Psi^\sharp) \in Post_\Pi(s)$ if and only if (i) there exists a product state $(\mathbf{r}, S, \Psi) \in Post(s)$ and (ii) $\Psi^\sharp(\mathbf{x}_{\mathcal{A}}) \equiv \bigwedge_{r \lhd (\mathbf{r}, S)} \bigwedge \{ \pi \in \Pi(r) \mid \Psi \to \pi \}$. In other words, the set of data valuations reachable by an abstract successor is the tightest over-approximation of the concrete set of reachable valuations, obtained as the conjunction of the available predicates from the predicate map that over-approximate this set.

Example 2 (*Continued from Ex. 1*) Consider the antichain from Fig. 2 (d). The concrete successors of $s = (\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1)$ are $(\langle q_1^1, q_1^2 \rangle, \{p_1\}, \Psi_1)$ and $(\langle q_1^1, q_1^2 \rangle, \{p_2\}, \Psi_2)$:

$$\Psi_1 \equiv \exists v', x', \Delta' \ . \ v' = 1 \wedge x = x' + 1 \wedge v = 1 \wedge \Delta = \Delta' \wedge 0 \leqslant x' < \Delta \wedge v = v',$$
$$\Psi_2 \equiv \exists v', x', \Delta' \ . \ v' = 1 \wedge x = x' + 1 \wedge v = 2 \wedge \Delta = \Delta' \wedge \Delta \leqslant x' < 2\Delta \wedge v = v' + 1.$$

With predicate map Π from Ex. 1, $Post_\Pi(s) = \{ (\langle q_1^1, q_1^2 \rangle, \{p_1\}, \Psi_1^\sharp), (\langle q_1^1, q_1^2 \rangle, \{p_2\}, \Psi_2^\sharp) \}$:

$$(\Psi_1 \to v = 1) \qquad\qquad\qquad \to (\Psi_1^\sharp \equiv v = 1),$$
$$(\Psi_2 \to v = 2 \text{ and } \Psi_2 \to \Delta < x) \to (\Psi_2^\sharp \equiv v = 2 \wedge \Delta < x). \qquad ∎$$

---

[11] A fact is a formula in Form$(\mathcal{D})$.

Finally, an *antichain tree* (or, simply antichain) $\mathcal{T}$ for $\mathcal{A}$ and B is a tree whose nodes are labeled with product states and whose edges are labeled by input symbols and concrete transition relations. Let $\mathbb{N}^*$ be the set of finite sequences of natural numbers that denote the positions in the tree. For a tree position $p \in \mathbb{N}^*$ and $i \in \mathbb{N}$, the position $p.i$ is a *child* of p. A set $S \subseteq \mathbb{N}^*$ is said to be *prefix-closed* if and only if, for each $p \in S$ and each prefix q of p, we have $q \in S$ as well. The root of the tree is denoted by the empty sequence ε.

Definition 1 Formally, an antichain $\mathcal{T}$ is a set of pairs $\langle s, p \rangle$, where s is a product state and $p \in \mathbb{N}^*$ is a tree position, such that (1) for each position $p \in \mathbb{N}^*$ there exists at most one product state s such that $\langle s, p \rangle \in \mathcal{T}$, (2) the set $\{p \mid \langle s, p \rangle \in \mathcal{T}\}$ is prefix-closed, (3) $(root_{\langle \mathcal{A}, B \rangle}, \varepsilon) \in \mathcal{T}$ where $root_{\langle \mathcal{A}, B \rangle} = (\langle \iota_1, \ldots, \iota_N \rangle, \{\iota_B\}, \top)$ is the label of the root, and (4) for each edge $(\langle s, p \rangle, \langle t, p.i \rangle)$ in $\mathcal{T}$, there exists a predicate map Π such that $t \in Post_\Pi(s)$. For the latter condition, if $s = (\mathbf{q}, P, \Phi)$ and $t = (\mathbf{r}, S, \Psi)$, there exists a unique rule $(\mathbf{q}, P) \xrightarrow{\sigma, \theta} (\mathbf{r}, S) \in \Delta^p$, and we shall sometimes denote the edge as $s \xrightarrow{\sigma, \theta} t$ or simply $s \xrightarrow{\theta} t$ when the tree positions or alphabet symbols are not important.

Each antichain node $n = (s, d_1 \ldots d_k) \in \mathcal{T}$ is naturally associated with a path from the root to itself $\rho \colon n_0 \xrightarrow{\sigma_1, \theta_1} n_1 \xrightarrow{\sigma_2, \theta_2} \ldots \xrightarrow{\sigma_k, \theta_k} n_k$. We denote by ρ_i the node n_i for each $i \in [0, k]$, and by $|\rho| = k$ the length of the path. The *path formula* associated with ρ is $\Theta(\rho) \equiv \bigwedge_{i=1}^k \theta_i(\mathbf{x}_{\mathcal{A}}^{i-1}, \mathbf{x}_{\mathcal{A}}^i)$ where $\mathbf{x}_{\mathcal{A}}^i = \{x^i \mid x \in \mathbf{x}_{\mathcal{A}}\}$ is a set of indexed variables for each $i \in [0, k]$.

Example 3 Consider the following path $\rho \colon (\langle q_0^1, q_0^2 \rangle, \{p_0\}, \top) \xrightarrow{\text{init}} (\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1) \xrightarrow{\mathbf{a_2}}$ $(\langle q_1^1, q_1^2 \rangle, \{p_2\}, \Delta < x) \xrightarrow{\mathbf{a_2}} (\langle q_1^1, q_1^2 \rangle, \varnothing, \Delta < x)$ in the antichain from Fig. 2 (c). The path formula of ρ is $\Theta(\rho) \equiv \theta_1 \wedge \theta_2 \wedge \theta_3$ where:

$\theta_1 \equiv v^1 = 1 \wedge x^1 = 0 \wedge 0 < \Delta^1$,
$\theta_2 \equiv v^2 = v^1 + 1 \wedge \Delta^2 = \Delta^1 \wedge v^2 = 2 \wedge x^2 = x^1 + 1 \wedge \Delta^1 \leqslant x^1 < 2\Delta^1 \wedge \neg(v^2 = v^1)$,
$\theta_3 \equiv v^3 = 2 \wedge \Delta^3 = \Delta^2 \wedge x^3 = x^2 + 1 \wedge \Delta_2 \leqslant x^2 < 2\Delta^2 \wedge \neg(v^3 = v^2)$. ∎

4.2 Counterexample-driven Abstraction Refinement

A *counterexample* is a path from the root of the antichain to a node which is labeled by an *accepting* product state. A product state $(\mathbf{q}, P, \Phi)$ is said to be *accepting* iff $(\mathbf{q}, P)$ is an accepting state of the product automaton $\mathcal{A}^e \times \overline{B}$, i.e., $\mathbf{q} \in F_{\mathcal{A}}$ and $P \cap F_B = \varnothing$. A counterexample is said to be *spurious* if its path formula is unsatisfiable, i.e., the path does not correspond to a concrete execution of $\mathcal{A}^e \times \overline{B}$. In this case, we need to (i) remove the path ρ from the current antichain and (ii) refine the abstract domain in order to exclude the occurrence of ρ from future state space exploration.

Let $\rho \colon root_{\langle \mathcal{A}, B \rangle} = (\mathbf{q}_0, P_0, \Phi_0) \xrightarrow{\theta_1} (\mathbf{q}_1, P_1, \Phi_1) \xrightarrow{\theta_2} \ldots \xrightarrow{\theta_k} (\mathbf{q}_k, P_k, \Phi_k)$ be a spurious counterexample in the following. For efficiency reasons, we would like to save as much work as possible and remove only the smallest suffix of ρ which caused the spuriousness. For some $j \in [0, k]$, let $\Theta^j(\rho) \equiv \Phi_j(\mathbf{x}_{\mathcal{A}}^0) \wedge \bigwedge_{i=j}^k \theta_i(\mathbf{x}_{\mathcal{A}}^{i-j}, \mathbf{x}_{\mathcal{A}}^{i-j+1})$ be the formula defining all sequences of data valuations that start in the set Φ_j and proceed along the suffix $(\mathbf{q}_j, P_j, \Phi_j) \to \ldots \to (\mathbf{q}_k, P_k, \Phi_k)$ of ρ. The *pivot* of a path ρ is the maximal position $j \in [0, k]$ such that $\Theta^j(\rho)$ is unsatisfiable, and -1 if ρ is not spurious.

Example 4 (*Continued from Ex. 3*) The path formula $\Theta(\rho) \equiv \theta_1 \wedge \theta_2 \wedge \theta_3$ from Ex. 3 is unsatisfiable, thus ρ is a spurious counterexample. Moreover, we have unsatisfiable $\Theta^1(\rho) \equiv \top \wedge \theta_2 \wedge \theta_3$ because of the unsatisfiable subformula $v^2 = 2 \wedge v^3 = 2 \wedge \neg(v^3 = v^2)$. Since $\Theta^2(\rho)$ is satisfiable, the pivot of ρ is 1. ∎

Finally, we describe the refinement of the predicate map, which ensures that a given spurious counterexample will never be found in a future iteration of the abstract state space exploration. The refinement is based on the notion of *interpolant* [25].

Definition 2 Given a formula $\Phi(\mathbf{x})$ and a sequence $\langle \theta_1(\mathbf{x}, \mathbf{x}'), \ldots, \theta_k(\mathbf{x}, \mathbf{x}') \rangle$ of formulae, an *interpolant* is a sequence of formulae $\mathbf{I} = \langle I_0(\mathbf{x}), \ldots, I_k(\mathbf{x}) \rangle$ where: (1) $\Phi \to I_0$, (2) $I_k \to \bot$, and (3) $I_{i-1}(\mathbf{x}) \wedge \theta_i(\mathbf{x}, \mathbf{x}') \to I_i(\mathbf{x}')$ for all $i \in [1, k]$.

Any given interpolant is a witness for the unsatisfiability of a (suffix) path formula $\Theta^j(\rho)$. Dually, if *Craig's Interpolation Lemma* [11] holds for the considered first-order data theory, any infeasible path formula is guaranteed to have an interpolant. The interpolant can be computed by means of Satisfiability Modulo Theories (SMT) solvers [9, 26].

Example 5 (Continued from Ex. 4) Let $\Phi \equiv \top$ (variables initially unconstraint) and the sequence of formula be $\langle \theta_1, \theta_2, \theta_3 \rangle$ from Ex. 3. An interpolant is a sequence $I = \langle \top, v = 2, \bot \rangle$. ∎

Given a spurious counterexample $\rho = (\mathbf{q}_0, P_0, \Phi_0) \xrightarrow{\theta_1} \ldots \xrightarrow{\theta_k} (\mathbf{q}_k, P_k, \Phi_k)$ with pivot $j \geqslant 0$, an interpolant $\mathbf{I} = \langle I_0, \ldots, I_{k-j} \rangle$ for the infeasible path formula $\Theta^j(\rho)$ can be used to refine the abstract domain by augmenting the predicate map Π. A simple possible refinement is to add the formula I_i into $\Pi((\mathbf{q}_{j+i}, P_{j+i}))$ for each $0 \leqslant i \leqslant (k - j)$. As an effect of this refinement, the antichain construction algorithm will avoid every path with the suffix $(\mathbf{q}_j, P_j, \Phi_j) \to \ldots \to (\mathbf{q}_k, P_k, \Phi_k)$ in a future iteration.

We use an improved version of this simple refinement in order to obtain more reusable predicates. If $I_i = C_i^1(\mathbf{y}_1) \wedge \ldots \wedge C_i^{m_i}(\mathbf{y}_{m_i})$ is a conjunctive normal form (CNF) of the i-th component of the interpolant, we consider the substate $(\mathbf{r}_i^\ell, S_i^\ell)$ for each $C_i^\ell(\mathbf{y}_\ell)$ where $\ell \in [1, m_i]$:

- $\mathbf{r}_i^\ell = \langle q_{i_1}, \ldots, q_{i_h} \rangle$ where $1 \leqslant i_1 < \ldots < i_h \leqslant N$ is the largest sequence of indices such that $\mathbf{x}_{i_g} \cap \mathbf{y}_\ell \neq \varnothing$ for each $g \in [1, h]$ and the set $\mathbf{x}_{i_g}$ of variables of the network component GRA A_{i_g},
- $S_i^\ell = P_i$ if $\mathbf{x}_B \cap \mathbf{y}_\ell \neq \varnothing$, and $S_i^\ell = \varnothing$, otherwise.

A predicate map Π is said to be *compatible* with a spurious path $\rho : s_0 \xrightarrow{\theta_1} \ldots \xrightarrow{\theta_k} s_k$ with pivot $j \geqslant 0$ if $s_j = (\mathbf{q}_j, P_j, \Phi_j)$ and there is an interpolant $\mathbf{I} = \langle I_0, \ldots, I_{k-j} \rangle$ of the suffix $\langle \theta_j, \ldots, \theta_k \rangle$ wrt. Φ_j such that, for each clause C of some equivalent CNF of I_i, $i \in [0, k-j]$, it holds that $C \in \Pi(r)$ for some substate $r \lhd s_{i+j}$. The following lemma proves that, under a predicate map compatible with a spurious path ρ, the antichain construction will exclude further paths that share the suffix of ρ starting with its pivot.

Lemma 3 *Let* $\rho : (\mathbf{q}_0, P_0, \Phi_0) \xrightarrow{\theta_0} (\mathbf{q}_1, P_1, \Phi_1) \xrightarrow{\theta_1} \ldots \xrightarrow{\theta_{k-1}} (\mathbf{q}_k, P_k, \Phi_k)$ *be a spurious counterexample and* Π *a predicate map compatible with* ρ. *Then, there is no sequence of product states* $(\mathbf{q}_j, P_j, \Psi_0), \ldots, (\mathbf{q}_k, P_k, \Psi_{k-j})$ *such that: (1)* $\Psi_0 \to \Phi_j$ *and (2)* $(\mathbf{q}_{i+1}, P_{i+1}, \Psi_{i-j+1}) \in Post_\Pi((\mathbf{q}_i, P_i, \Psi_{i-j}))$ *for all* $i \in [j, k-1]$.

Proof Let $j \in [0, k]$ be the pivot of ρ. Since ρ is spurious, there exists an interpolant $\mathbf{I} = \langle I_0, \ldots, I_{k-j} \rangle$ for Φ_j and $\langle \theta_j, \ldots, \theta_k \rangle$. It is sufficient to prove that $\Psi_i \to I_i$ for all $i \in [0, k-j]$. Since $I_{k-j} = \bot$, we obtain $\Psi_{k-j} = \bot$, and consequently $(\mathbf{q}_{k-j}, P_{k-j}, \bot) \in Post_\Pi((\mathbf{q}_{k-j-1}, P_{k-j-1}, \Psi_{k-j-1}))$. By the definition of $Post_\Pi$, we have $(\mathbf{q}_{k-j}, P_{k-j}, \bot) \in Post((\mathbf{q}_{k-j-1}, P_{k-j-1}, \Psi_{k-j-1}))$, which contradicts with the definition of $Post$. We show that $\Psi_i \to I_i$ for all $i \in [0, k-j]$, by induction on $k - j$. For the base case $k - j = 0$, we have

$\Psi_0 \to \Phi_j \to I_0$. For the induction step, we assume $\Psi_i \to I_i$ for all $i \in [0, k-j-1]$ and prove $\Psi_{k-j} \to I_{k-j}$. By the induction hypothesis, we have:

$$\Psi_{k-j-1}(\mathbf{x}_{\mathcal{A}}) \to I_{k-j-1}(\mathbf{x}_{\mathcal{A}}) \text{ and}$$
$$\Psi_{k-j-1}(\mathbf{x}_{\mathcal{A}}) \wedge \theta_{k-j-1}(\mathbf{x}_{\mathcal{A}}, \mathbf{x}'_{\mathcal{A}}) \to I_{k-j-1}(\mathbf{x}_{\mathcal{A}}) \wedge \theta_{k-j-1}(\mathbf{x}_{\mathcal{A}}, \mathbf{x}'_{\mathcal{A}}) \to I_{k-j}(\mathbf{x}'_{\mathcal{A}}).$$

Let $C_1 \wedge \ldots \wedge C_\ell$ be the CNF of I_{k-j}. Since Π is compatible with ρ, for each clause C_i, there exists a substate $r \lhd (\mathbf{q}_k, P_k)$ such that $C_i \in \Pi(r)$. By the definition of $Post_\Pi$, we obtain that $\Psi_{k-j} \to C_i$ for each $i \in [1, \ell]$, hence $\Psi_{k-j} \to I_{k-j}$. $\square$

Observe that the refinement induced by interpolation is *local* since Π associates sets of predicates with substates of the states in $\mathcal{A}^e \times \overline{B}$, and the update impacts only the states occurring within the suffix of that particular spurious counterexample.

4.3 Subsumption

The main optimization of antichain-based algorithms [1] for checking language inclusion of automata over finite alphabets is that product states that are *subsets* of already visited states are never stored in the antichain. On the other hand, language emptiness semi-algorithms, based on *predicate abstraction* [25] use a similar notion to cover newly generated abstract successor states by those that were visited sooner and that represent larger sets of configurations. In this case, state coverage does not only increase efficiency but also ensures termination of the semi-algorithm in many practical cases.

In this section, we generalize the subset relation used in classical antichain algorithms with the notion of coverage from predicate abstraction, and we define a more general notion of *subsumption* for generic register automata. Given a state $(\mathbf{q}, P)$ of the product automaton $\mathcal{A}^e \times \overline{B}$ and a valuation $\nu \in \mathcal{D}^{\mathbf{x}_{\mathcal{A}}}$, the *residual language* $\mathcal{L}_{(\mathbf{q}, P, \nu)}(\mathcal{A}^e \times \overline{B})$ is the set of traces w accepted by $\mathcal{A}^e \times \overline{B}$ from the state $(\mathbf{q}, P)$ such that ν is the first valuation which occurs on w. This notion is then lifted to a product state $s = (\mathbf{q}, P, \Phi)$ as follows: $\mathcal{L}_s(\mathcal{A}^e \times \overline{B}) = \bigcup_{(\mathbf{q}, P, \nu) \in [[s]]} \mathcal{L}_{(\mathbf{q}, P, \nu)}(\mathcal{A}^e \times \overline{B})$ where $[[s]] = \{(\mathbf{q}, P, \nu) \mid \nu \models \Phi\}$—i.e. the set of configurations of the product automaton $\mathcal{A}^e \times \overline{B}$ represented by the given product state s.

Definition 3 Given a GRAN $\mathcal{A}$ and a GRA B, a partial order $\sqsubseteq$ is a *subsumption* provided that, for any two product states s and t, we have $s \sqsubseteq t$ only if $\mathcal{L}_s(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_t(\mathcal{A}^e \times \overline{B})$.

A procedure for checking the emptiness of $\mathcal{A}^e \times \overline{B}$ needs not continue the search from a product state s if it has already visited a product state t that subsumes s. The intuition is that any counterexample discovered from s can also be discovered from t. The trace inclusion semi-algorithm described below in Section 4.4 works, in principle, with any given subsumption relation. In practice, our implementation uses the subsumption relation defined by the lemma below:

Lemma 4 *The relation defined such that* $(\mathbf{q}, P, \Phi) \sqsubseteq_{img} (\mathbf{r}, S, \Psi) \iff \mathbf{q} = \mathbf{r}, P \supseteq S, \text{ and } \Phi \to \Psi$ *is a subsumption.*

Proof For any valuation $\nu \in \mathcal{D}^{\mathbf{x}_{\mathcal{A}}}$, we have $\mathcal{L}_{(\mathbf{q}, P, \nu)}(\mathcal{A}^e \times \overline{B}) = \mathcal{L}_{(\mathbf{q}, \nu)}(\mathcal{A}^e) \cap \mathcal{L}_{(P, \nu)}(\overline{B})$. Since $P \supseteq S$, we have $\mathcal{L}_{(P, \nu)}(B) \supseteq \mathcal{L}_{(S, \nu)}(B)$, thus $\mathcal{L}_{(P, \nu)}(\overline{B}) \subseteq \mathcal{L}_{(S, \nu)}(\overline{B})$. We obtain that $\mathcal{L}_{(\mathbf{q}, P, \nu)}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_{(\mathbf{r}, \nu)}(\mathcal{A}^e) \cap \mathcal{L}_{(S, \nu)}(\overline{B}) = \mathcal{L}_{(\mathbf{r}, S, \nu)}(\mathcal{A}^e \times \overline{B})$. Since moreover $\Phi \to \Psi$, we have that $\mathcal{L}_{(\mathbf{q}, P, \Phi)}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_{(\mathbf{r}, S, \Phi)}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_{(\mathbf{r}, S, \Psi)}(\mathcal{A}^e \times \overline{B})$. $\square$

Algorithm 1 Trace Inclusion Semi-algorithm

---

input:
    1. A GRAN $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ such that $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$ for all $i \in [1, N]$.
    2. A GRA $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ such that $\mathbf{x}_B \subseteq \bigcup_{i=1}^{N} \mathbf{x}_i$.
  output: True if $\mathcal{L}(\mathcal{A}) \downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$, otherwise a trace $\tau \in \mathcal{L}(\mathcal{A}) \downarrow_{\mathbf{x}_B} \setminus \mathcal{L}(B)$.

1: $\Pi \leftarrow \varnothing$, Visited $\leftarrow \varnothing$, Next $\leftarrow \langle root_{\langle \mathcal{A}, B \rangle}, \varepsilon \rangle$, Subsume $\leftarrow \varnothing$
2: **while** Next $\neq \varnothing$ **do**
3:      choose curr $\in$ Next and move curr from Next to Visited
4:      match curr with $\langle s, p \rangle$
5:      **if** s is an accepting product state **then**
6:          let ρ be the path from the root to curr and k be the pivot of ρ
7:          **if** $k \geqslant 0$ **then**
8:              $\Pi \leftarrow$ REFINEPREDICATEMAPBYINTERPOLATION(Π, ρ, k)
9:              rem $\leftarrow$ SUBTREE(ρ_k)
10:             **for** $(n, m) \in$ Subsume such that $m \in$ rem **do**
11:                move n from Visited to Next
12:             remove rem from (Visited, Next, Subsume)
13:             add ρ_k to Next
14:          **else**
15:              return EXTRACTCOUNTEREXAMPLE(ρ)
16:      **else**
17:          $i \leftarrow 0$
18:          **for** $t \in Post_\Pi(s)$ **do**
19:             **if** there exists $m = \langle t', p' \rangle \in$ Visited such that $t \sqsubseteq t'$ **then**
20:                add (curr, m) to Subsume
21:             **else**
22:                rem $\leftarrow \{n \in$ Next $\mid n = \langle t', p' \rangle$ and $t' \sqsubset t\}$
23:                succ $\leftarrow \langle t, p.i \rangle$
24:                $i \leftarrow i + 1$
25:                **for** $n \in$ Visited such that n has a successor $m \in$ rem **do**
26:                  add (n, succ) to Subsume
27:                **for** $(n, m) \in$ Subsume such that $m \in$ rem **do**
28:                  add (n, succ) to Subsume
29:                remove rem from (Visited, Next, Subsume)
30:                add succ to Next
31: return *True*

Example 6 In the antichain from Fig. 2 (d), $(\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1) \sqsubseteq_{img} (\langle q_1^1, q_1^2 \rangle, \{p_1\}, v = 1)$ because $\langle q_1^1, q_1^2 \rangle = \langle q_1^1, q_1^2 \rangle$, $\{p_1\} \supseteq \{p_1\}$, and $v = 1 \rightarrow v = 1$. ∎

The language inclusion algorithm for non-deterministic automata on finite alphabets [1] uses also a more sophisticated subsumption relation based on a pre-computed simulation [27] between the states of the automata. We have defined a similar notion of simulation for generic register automata and an algorithm for computing such simulations. Details concerning data simulations and their integration within the framework of antichain-based abstraction refinement are described in Section 5.

4.4 The Trace Inclusion Semi-algorithm

With the previous definitions, Algorithm 1 describes the procedure for checking trace inclusion. It uses a classical worklist iteration loop (lines 2-30) that builds an antichain tree by simultaneously unfolding the expansion $\mathcal{A}^e$ of the network $\mathcal{A}$ and the complement $\overline{B}$ of the the observer B, while searching for a counterexample trace $w \in \mathcal{L}(\mathcal{A}^e \times \overline{B})$. Both $\mathcal{A}^e$ and $\overline{B}$ are built on-the-fly, during the abstract state space exploration.

Within Algorithm 1, the antichain is represented as a set of nodes. Each node is a tuple $\langle s, p \rangle$ where s is a product state and p is a position in the tree. The processed antichain nodes are kept in the set Visited, and their abstract successors, not yet processed, are kept in the

set Next. Initially, Visited $= \varnothing$ and Next $= \{\langle root_{\mathcal{A},B}, \varepsilon \rangle\}$. The algorithm uses a predicate map Π, which is initially empty (line 1).

We keep a set of subsumption edges Subsume $\subseteq$ Visited $\times$ (Visited $\cup$ Next) with the following meaning: $(\langle s, p \rangle, \langle t, q \rangle) \in$ Subsume for two antichain nodes, where s, t are product states and $p, q \in \mathbb{N}^*$ are tree positions, if and only if there exists an abstract successor $s' \in Post_{\Pi}(s)$ such that $s' \sqsubseteq t$ (Definition 3). Observe that we do not explicitly store a subsumed successor of a product state s from the antichain; instead, we add a subsumption edge between the node labeled with s and the node that subsumes that particular successor. The algorithm terminates when each abstract successor of a node from Next is subsumed by some node from Visited.

An iteration of Algorithm 1 starts by choosing a current antichain node curr $= \langle s, p \rangle$ from Next and moving it to Visited (line 3). If the product state s is accepting (line 5), we check the counterexample path ρ, from the root of the antichain to curr, for spuriousness, by computing its pivot k (see Section 4.2). If $k \geq 0$, then ρ is a spurious counterexample (line 7), and the path formula of the suffix of ρ, which starts with position k, is infeasible. In this case, we compute an interpolant for the suffix and refine the current predicate map Π by adding the predicates from the interpolant to the corresponding substates of the product states from the suffix (line 8).

The function REFINEPREDICATEMAPBYINTERPOLATION updates the predicate map using the principle described in Section 4.2. Subsequently, we remove (line 12) from the current antichain the subtree rooted at the pivot node ρ_k, i.e., the k-th node on the path ρ (line 9), and add ρ_k to Next in order to trigger a recomputation of this subtree with the new predicate map. Moreover, all nodes with a successor previously subsumed by a node in the removed subtree are moved from Visited back to Next in order to reprocess them (line 11).

On the other hand, if the counterexample ρ is found to be real ($k = -1$), any valuation $\nu \in \bigcup_{i=0}^{|\rho|} \mathcal{D}^{\mathbf{x}_{\mathcal{A}}^i}$ that satisfies the path formula $\Theta(\rho)$ yields a counterexample trace $w \in \mathcal{L}(\mathcal{A}) \downarrow_{\mathbf{x}_B} \backslash \mathcal{L}(B)$, obtained by ignoring all variables from $\mathbf{x}_{\mathcal{A}} \backslash \mathbf{x}_B$ (line 15).

If the current node is not accepting, we generate its abstract successors (line 18). In order to keep in the antichain only nodes that are incomparable wrt the subsumption relation $\sqsubseteq$, we add a successor t of s to Next (lines 23 and 30) only if it is not subsumed by another product state from a node $m \in$ Visited. Otherwise, we add a subsumption edge (curr, m) to the set Subsume (line 20). Furthermore, if t is not subsumed by another state in Visited, we remove from Next all nodes $\langle t', p' \rangle$ such that t strictly subsumes t' (lines 22 and 29) and add subsumption edges to the node storing t from all nodes with a removed successor (line 26) or a removed subsumption edge (line 28).

The following theorem states the soundness of our trace inclusion semi-algorithm.

Theorem 1 *Let $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ be a GRAN such that $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$ for all $i \in [1, N]$, and let $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ be a GRA such that $\mathbf{x}_B \subseteq \bigcup_{i=1}^{N} \mathbf{x}_i$. If Algorithm 1 terminates and returns true on input $\mathcal{A}$ and B, then $\mathcal{L}(\mathcal{A}) \downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$.*

The dual question "if there exists a counterexample trace $w \in \mathcal{L}(\mathcal{A}) \downarrow_{\mathbf{x}_B} \backslash \mathcal{L}(B)$, will Algorithm 1 discover it?" can also be answered positively, using an implementation that enumerates the abstract paths in a systematic way, e.g., by using a breadth-first path exploration. This can be done using a queue to implement the Next set in Algorithm 1.

4.5 Proof of Theorem 1

Given a network $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ where $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$ for all $i \in [1, N]$ and an observer $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$, we recall that a configuration of the product automaton $\mathcal{A}^e \times \overline{B}$ is a tuple $(\langle q_1, \ldots, q_N \rangle, P, \nu) \in Q_1 \times \ldots \times Q_N \times 2^{Q_B} \times \mathcal{D}^{\mathbf{x}_{\mathcal{A}}}$, and a node of the antichain $\mathcal{T}$ is a pair $\langle s, p \rangle$ where s is a product state for $\mathcal{A}$ and B and $p \in \mathbb{N}^*$ is a tree position. Moreover, $root_{\langle \mathcal{A}, B \rangle} = (\langle \iota_1, \ldots, \iota_N \rangle, \{\iota_B\}, \top)$ is the product state that labels the root of $\mathcal{T}$. In the following, let $\Gamma = (\Pi, \mathtt{Visited}, \mathtt{Next}, \mathtt{Subsume})$ be an *antichain state* where Π is the predicate map, and $\mathtt{Visited}$, $\mathtt{Next}$, and $\mathtt{Subsume}$ are the sets of antichain nodes handled by Algorithm 1.

We say that Γ is a *closed antichain state* if and only if, for all nodes $\langle s, p \rangle \in \mathtt{Visited}$ and every successor $(\mathbf{q}, P, \nu) \in succ_{\mathcal{A}^e \times \overline{B}}([\![s]\!])$ of a configuration of the product automaton $\mathcal{A}^e \times \overline{B}$ represented by the product state s, there exists a node $\langle t, r \rangle \in \mathtt{Visited} \cup \mathtt{Next}$ such that $\mathcal{L}_{(\mathbf{q}, P, \nu)}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_t(\mathcal{A}^e \times \overline{B})$ and one of the following holds:

- $r = p.i$ for some $i \in \mathbb{N}$, i.e., $\langle t, r \rangle$ is a child of $\langle s, p \rangle$ in the antichain $\mathcal{T} = \mathtt{Visited} \cup \mathtt{Next}$, or
- $(\langle s, p \rangle, \langle t, r \rangle) \in \mathtt{Subsume}$.

In other words, the current antichain $\mathcal{T}$, defined as the union of the sets $\mathtt{Visited}$ and $\mathtt{Next}$, is in a closed state if the residual language of every successor of a configuration of the product automaton $\mathcal{A}^e \times \overline{B}$ that is covered by a visited product state must be included in the residual language of a product state stored in the antichain, either as a direct successor in the tree or via a subsumption edge.

For a product state s, we define $Dist(s) = \min \{|w| \mid w \in \mathcal{L}_s(\mathcal{A}^e \times \overline{B})\}$, and $Dist(s) = \infty$ if and only if $\mathcal{L}_s(\mathcal{A}^e \times \overline{B}) = \varnothing$. For a finite non-empty set of antichain nodes S, we define $Dist(S) = \min \{Dist(s) \mid \langle s, p \rangle \in S\}$ with $Dist(\varnothing) = \infty$.

We now prove several auxiliary lemmas.

Lemma 5 *Given a network $\mathcal{A}$ and an observer B, for any product state s of $\mathcal{A}$ and B, we have $succ_{\mathcal{A}^e \times \overline{B}}([\![s]\!]) = \bigcup_{t \in Post(s)} [\![t]\!]$.*

Proof Let $s = (\mathbf{q}, P, \Phi)$. "$\subseteq$" Let $(\mathbf{r}, S, \mu) \in succ_{\mathcal{A}^e \times \overline{B}}([\![s]\!])$ be a configuration of $\mathcal{A}^e \times \overline{B}$ for which there exists $(\mathbf{q}, P, \nu) \in [\![s]\!]$ such that $(\mathbf{q}, P, \nu) \xrightarrow{\sigma, \theta} (\mathbf{r}, S, \mu)$. Then there exists a unique rule $(\mathbf{q}, P) \xrightarrow{\sigma, \theta} (\mathbf{r}, S) \in \Delta^p$ such that $(\nu, \mu) \models \theta$. Moreover, if $(\mathbf{q}, P, \nu) \in [\![s]\!]$, we have $\nu \models \Phi$. Let $t = (\mathbf{r}, S, \Psi) \in Post(s)$ where $\Psi(\mathbf{x}_{\mathcal{A}}) \equiv \exists \mathbf{x}'_{\mathcal{A}} . \Phi(\mathbf{x}'_{\mathcal{A}}) \wedge \theta(\mathbf{x}'_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}})$. We have $\mu \models \Psi$, hence $(\mathbf{r}, S, \mu) \in [\![t]\!]$. "$\supseteq$" Let $(\mathbf{r}, S, \mu) \in [\![t]\!]$ for some $t \in Post(s)$. Then we have $t = (\mathbf{r}, S, \Psi)$ where $\Psi(\mathbf{x}_{\mathcal{A}}) \equiv \exists \mathbf{x}'_{\mathcal{A}} . \Phi(\mathbf{x}'_{\mathcal{A}}) \wedge \theta(\mathbf{x}'_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}})$. Since $\mu \models \Psi$, there exists $\nu \models \Phi$ such that $(\mathbf{q}, P, \nu) \xrightarrow{\sigma, \theta} (\mathbf{r}, S, \mu)$. Hence $(\mathbf{q}, P, \nu) \in [\![s]\!]$, thus $(\mathbf{r}, S, \mu) \in succ_{\mathcal{A}^e \times \overline{B}}([\![s]\!])$. $\square$

Lemma 6 *Given a network $\mathcal{A}$, an observer B, and a predicate map Π, for any product state s of $\mathcal{A}^e \times \overline{B}$ and any product state $t \in Post(s)$, there exists $t' \in Post_{\Pi}(s)$ such that $[\![t]\!] \subseteq [\![t']\!]$.*

Proof Let $t = (\mathbf{r}, S, \Psi) \in Post(s)$. By the definition of $Post_{\Pi}$, we have $t' = (\mathbf{r}, S, \Psi^{\sharp}) \in Post_{\Pi}(s)$, where $\Psi \rightarrow \Psi^{\sharp}$, thus $[\![t]\!] \subseteq [\![t']\!]$. $\square$

Lemma 7 *Given a network $\mathcal{A}$, an observer B, and a predicate map Π, for each product state s and each configuration $(\mathbf{q}, P, \nu) \in succ_{\mathcal{A}^e \times \overline{B}}([\![s]\!])$ there exists a product state $t \in Post_{\Pi}(s)$ such that $(\mathbf{q}, P, \nu) \in [\![t]\!]$.*

Proof We use the fact that $succ_{\mathcal{A}^e \times \overline{B}}(\llbracket s \rrbracket) = \bigcup_{t \in Post(s)} \llbracket t \rrbracket$ (Lemma 5) and that for each $t \in Post(s)$ there exists $t' \in Post_{\Pi}(s)$ such that $\llbracket t \rrbracket \subseteq \llbracket t' \rrbracket$ (Lemma 6). $\qquad\square$

The proof of soundness of Algorithm 1 relies on the inductive invariants (Inv_1) and (Inv_2) from the following lemma.

Lemma 8 *The following invariants hold each time line 2 is reached in Algorithm 1:*
- *(Inv_1)* $\Gamma = (\Pi, \texttt{Visited}, \texttt{Next}, \texttt{Subsume})$ *is closed,*
- *(Inv_2)* $Dist(root_{\langle \mathcal{A}, B \rangle}) < \infty \rightarrow Dist(\texttt{Visited}) > Dist(\texttt{Next})$.

Proof Initially, when coming to line 2 for the first time, we have $\texttt{Visited} = \varnothing$, thus $Dist(\texttt{Visited}) = \infty$, and both invariants hold trivially. For the case when coming to line 2 after executing the loop body, we denote by:

$$\Gamma_{old} = (\Pi_{old}, \texttt{Visited}_{old}, \texttt{Next}_{old}, \texttt{Subsume}_{old}) \text{ and}$$
$$\Gamma_{new} = (\Pi_{new}, \texttt{Visited}_{new}, \texttt{Next}_{new}, \texttt{Subsume}_{new})$$

the antichain states before and after the execution of the main loop. We assume that both invariants hold for Γ_{old}.

(Inv_1) Let $\langle s, p \rangle \in \texttt{Visited}_{new}$ and $(\mathbf{q}, P, \mathsf{v}) \in succ_{\mathcal{A}^e \times \overline{B}}(\llbracket s \rrbracket)$. We distinguish two cases according to the control path taken inside the main loop:

(1) If the test on line 5 is positive, the predicate map is augmented, i.e., $\Pi_{new} \supseteq \Pi_{old}$ (line 8). Let $\Gamma' = (\Pi_{new}, \texttt{Visited}_{old}, \texttt{Next}_{old}, \texttt{Subsume}_{old})$ be the next antichain state. Clearly Γ' is closed provided that Γ_{old} is. Next, let $n_{pivot} \in \texttt{Visited}_{old}$ be the pivot of the path to the current node (line 6) and define the following sets of nodes:

$$T = \textsc{SubTree}(n_{pivot}),$$
$$S = \{n \in \texttt{Visited}_{old} \mid \exists m \in T . (n, m) \in \texttt{Subsume}_{old}\}.$$

Then we obtain (lines 10–13):

$$\texttt{Visited}_{new} = \texttt{Visited}_{old} \backslash (S \cup T),$$
$$\texttt{Next}_{new} = ((\texttt{Next}_{old} \cup S) \backslash T) \cup \{n_{pivot}\},$$
$$\texttt{Visited}_{new} \cup \texttt{Next}_{new} = ((\texttt{Visited}_{old} \cup \texttt{Next}_{old}) \backslash T) \cup \{n_{pivot}\}.$$

Since Γ' is closed, there exists a node $\langle t, r \rangle \in \texttt{Visited}_{old} \cup \texttt{Next}_{old}$ such that $L_{(\mathbf{q}, P, \mathsf{v})}(\mathcal{A}^e \times \overline{B}) \subseteq L_t(\mathcal{A}^e \times \overline{B})$ and either $r = p.i$ for some $i \in \mathbb{N}$ or $(\langle s, p \rangle, \langle t, r \rangle) \in \texttt{Subsume}_{old}$. We distinguish two cases:

(a) $\langle t, r \rangle \notin T$. Then $\langle t, r \rangle \in \texttt{Visited}_{new} \cup \texttt{Next}_{new}$ and, because $\texttt{Subsume}_{new} = \texttt{Subsume}_{old} \cap (\texttt{Visited}_{new} \times (\texttt{Visited}_{new} \cup \texttt{Next}_{new}))$, we obtain that Γ_{new} is closed as well.

(b) $\langle t, r \rangle \in T$. Then we distinguish two further cases:
  (i) If $r = p.i$ for some $i \in \mathbb{N}$, since we have assumed that $\langle s, p \rangle \in \texttt{Visited}_{new}$, we have $\langle s, p \rangle \notin T$. The only possibility is then $\langle t, r \rangle = n_{pivot}$ and $\langle s, p \rangle$ is the parent of n_{pivot}. In this case, we have $\langle t, r \rangle \in \texttt{Next}_{new}$.
  (ii) If $(\langle s, p \rangle, \langle t, r \rangle) \in \texttt{Subsume}_{old}$, then $\langle s, p \rangle \in S$, which contradicts the assumption $\langle s, p \rangle \in \texttt{Visited}_{new}$.

(2) Otherwise, the test on line 5 is negative, in which case we have $\Pi_{new} = \Pi_{old}$ and $\texttt{Visited}_{new} = \texttt{Visited}_{old} \cup \{\texttt{curr}\}$. For each $(\mathbf{q}, P, \mathsf{v}) \in succ_{\mathcal{A}^e \times \overline{B}}(\llbracket s \rrbracket)$ there exists $t \in Post_{\Pi}(s)$ such that $L_{(\mathbf{q}, P, \mathsf{v})}(\mathcal{A}^e \times \overline{B}) \subseteq L_t(\mathcal{A}^e \times \overline{B})$ (by Lemma 7). We distinguish two cases:

(a) $\langle s,p\rangle = \mathtt{curr}$. In this case, either (i) there is $\langle t',p'\rangle \in \mathtt{Visited}_{old}$ such that $t \sqsubseteq t'$, and then we also have $\mathcal{L}_{(\mathbf{q},P,\mathbf{v})}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_{t'}(\mathcal{A}^e \times \overline{B})$ (Definition 3) and $(\langle s,p\rangle,\langle t',p'\rangle) \in \mathtt{Subsume}_{new}$ (added on line 20), or (ii) $(t,p.i) \in \mathtt{Next}_{new}$ for some $i \in \mathbb{N}$ (added on lines 23 and 30).

(b) Otherwise $\langle s,p\rangle \in \mathtt{Visited}_{old}$. As Γ' is closed, there is $\langle u,r\rangle \in \mathtt{Visited}_{old} \cup \mathtt{Next}_{old}$ such that $\mathcal{L}_{(\mathbf{q},P,\mathbf{v})}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_u(\mathcal{A}^e \times \overline{B})$ and either $r = p.i$ for some $i \in \mathbb{N}$ or $(\langle s,p\rangle,\langle u,r\rangle) \in \mathtt{Subsume}_{old}$. We distinguish two sub-cases:

  (i) $\langle u,r\rangle \in \mathtt{rem}$ (line 22). Then $\mathcal{L}_u(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_t(\mathcal{A}^e \times \overline{B})$ (Definition 3), hence $\mathcal{L}_{(\mathbf{q},P,\mathbf{v})}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_t(\mathcal{A}^e \times \overline{B})$. If $r = p.i$, then $(\langle s,p\rangle,\langle t,r'\rangle) \in \mathtt{Subsume}_{new}$ for some $r' \in \mathbb{N}^*$ (added on line 26). Else, if $(\langle s,p\rangle,\langle u,r\rangle) \in \mathtt{Subsume}_{old}$, we have $(\langle s,p\rangle,\langle t,r'\rangle) \in \mathtt{Subsume}_{new}$ for some $r' \in \mathbb{N}^*$ (added on line 28). In both cases, we obtain that Γ_{new} is closed.

  (ii) $\langle u,r\rangle \notin \mathtt{rem}$. Then $\langle u,r\rangle \in \mathtt{Visited}_{new} \cup \mathtt{Next}_{new}$. Since $\mathtt{Subsume}_{new} = \mathtt{Subsume}_{old} \cap (\mathtt{Visited}_{new} \times (\mathtt{Visited}_{new} \cup \mathtt{Next}_{new}))$, we obtain that Γ_{new} is closed.

(Inv_2) We distinguish two cases:

1. If $Dist(\mathtt{Visited}_{new}) = \infty$, it is sufficient to show that $Dist(\mathtt{Next}_{new}) < \infty$. Suppose, by contradiction, that $Dist(\mathtt{Next}_{new}) = \infty$, hence $Dist(\mathtt{Visited}_{new} \cup \mathtt{Next}_{new}) = \infty$, and since $root_{\langle \mathcal{A},B\rangle} \in \mathtt{Visited}_{new} \cup \mathtt{Next}_{new}$, we obtain $Dist(root_{\langle \mathcal{A},B\rangle}) = \infty$, contradiction.

2. Otherwise, $Dist(\mathtt{Visited}_{new}) < \infty$ and there exists a node $\langle s,p\rangle \in \mathtt{Visited}_{new}$ such that $Dist(\mathtt{Visited}_{new}) = Dist(s) < \infty$. Let $w = (\mathbf{v}_0,\sigma_0),(\mathbf{v}_1,\sigma_1),\ldots,(\mathbf{v}_n,\diamond) \in \mathcal{L}_s(\mathcal{A}^e \times \overline{B})$ be a trace such that $Dist(\mathtt{Visited}_{new}) = n$. Then there exists a run $(\mathbf{q}_0,P_0,\mathbf{v}_0) \xrightarrow{\sigma_0} (\mathbf{q}_1,P_1,\mathbf{v}_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-1}} (\mathbf{q}_n,P_n,\mathbf{v}_n)$ of $\mathcal{A}^e \times \overline{B}$ over w such that $(\mathbf{q}_0,P_0,\mathbf{v}_0) \in [\![s]\!]$ and $(\mathbf{q}_n,P_n)$ a final state of $\mathcal{A}^e \times \overline{B}$. Since Γ_{new} is closed due to (Inv_1) and $(\mathbf{q}_1,P_1,\mathbf{v}_1) \in succ_{\mathcal{A}^e \times \overline{B}}([\![s]\!])$, there exists a node $\langle s_1,p_1\rangle \in \mathtt{Visited}_{new} \cup \mathtt{Next}_{new}$ such that $\mathcal{L}_{(\mathbf{q}_1,P_1,\mathbf{v}_1)}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_{s_1}(\mathcal{A}^e \times \overline{B})$. If $\langle s_1,p_1\rangle \in \mathtt{Next}_{new}$, we obtain that $Dist(\mathtt{Next}_{new}) \leqslant n-1$, and we are done. Otherwise, $\langle s_1,p_1\rangle \in \mathtt{Visited}_{new}$, and we can repeat the same argument inductively, to discover a sequence of nodes $\langle s_1,p_1\rangle,\ldots,\langle s_n,p_n\rangle \in \mathtt{Visited}_{new}$ such that $\mathcal{L}_{(\mathbf{q}_i,P_i,\mathbf{v}_i)}(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_{s_n}(\mathcal{A}^e \times \overline{B})$ for all $i \in [1,n]$. Since $(\mathbf{q}_n,P_n)$ is a final state of $\mathcal{A}^e \times \overline{B}$, we have $(\mathbf{v}_n,\diamond) \in \mathcal{L}_{(\mathbf{q}_i,P_i,\mathbf{v}_i)}(\mathcal{A}^e \times \overline{B})$, thus $(\mathbf{v}_n,\diamond) \in \mathcal{L}_{s_n}(\mathcal{A}^e \times \overline{B})$, and s_n is an accepting product state. But this contradicts with the fact that accepting product states are never stored in the antichain. $\qquad\square$

With the above lemmas at hand, we can finally prove Theorem 1:

Proof If Algorithm 1 terminates and reports true, this is because $\mathtt{Next} = \varnothing$, hence $Dist(\mathtt{Next}) = \infty$. By Lemma 8 ($Inv_2$), we obtain that $Dist(root_{\langle \mathcal{A},B\rangle}) = \infty$. Suppose, by contradiction, that $\mathcal{L}(\mathcal{A})\!\downarrow_{\mathbf{x}_B} \not\subseteq \mathcal{L}(B)$. By Lemma 2, there exists a trace

$$w = (\mathbf{v}_0,\sigma_0)(\mathbf{v}_1,\sigma_1)\ldots(\mathbf{v}_n,\diamond) \in \mathcal{L}(\mathcal{A}^e \times \overline{B}).$$

Thus we have a run of $\mathcal{A}^e \times \overline{B}$ over w:

$$(\mathbf{q}_0,P_0,\mathbf{v}_0) \xrightarrow{\sigma_0} (\mathbf{q}_1,P_1,\mathbf{v}_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-1}} (\mathbf{q}_n,P_n,\mathbf{v}_n)$$

where $\mathbf{q}_0 = \langle \iota_1,\ldots,\iota_N\rangle$, $P_0 = \{\iota_B\}$, $\mathbf{q}_n$ is final in $\mathcal{A}^e$, $P_n \cap F_B = \varnothing$. But, since $(\mathbf{q}_0,P_0,\mathbf{v}_0) \in [\![root_{\langle \mathcal{A},B\rangle}]\!]$, we have $w \in \mathcal{L}_{root_{\langle \mathcal{A},B\rangle}}(\mathcal{A}^e \times \overline{B})$. Hence, $Dist(root_{\langle \mathcal{A},B\rangle}) \leqslant n$, which is in contradiction with the fact that $Dist(root_{\langle \mathcal{A},B\rangle}) = \infty$. Consequently, it must be the case that $\mathcal{L}(\mathcal{A})\!\downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$. $\qquad\square$

5 Simulations on Generic Register Automata

In the classical setting of finite state automata over finite alphabets, a *simulation* [27] is a relation on the states of an automaton which is invariant with respect to its transition relation. The simulation-based approach to checking language inclusion between two automata A and B first computes a simulation relation on the union of the states of A and B, and then checks whether the pair of initial states is a member of the simulation relation. Note that this is not a complete decision procedure for language inclusion, because there exist automata such that $\mathcal{L}(A) \subseteq \mathcal{L}(B)$, but the initial state of A is not simulated by the initial state of B. However, a pre-computed simulation relation can be used to speed up the convergence of the antichain-based method, by weakening (i.e. generalizing) the subsumption relation used by the antichain construction algorithm [1]. In practice, the experimental evaluation in [1] shows a significant improvement of running times, when simulations are used.

In the below subsection, we first introduce a concept of *data simulations* suitable for GRAs, together with an algorithm that computes useful under-approximations of the largest data simulation on a given GRA. In the next subsection, we then propose a way of using data simulations to enhance the convergence of Algorithm 1 between a GRAN and a GRA in a similar way as classical simulations are integrated with the antichain-based language inclusion algorithm for automata over finite alphabets [1].

We note that, in the context of classical automata, an approach going beyond the combination of antichains and simulation relations has been proposed [7]. It is based on using *congruence relations* instead of antichains. However, their usage in the context of generic register automata is so far unclear, and we leave it as an interesting subject for future work.

5.1 Data Simulations and Their Computation

Our notion of data simulations is defined as follows.

Definition 4 A relation $R \subseteq Q \times \mathcal{D}^{\mathbf{x}} \times Q$ is a *data simulation* for a GRA $A = \langle \Sigma, \mathcal{D}, \mathbf{x}, Q, \iota, F, \Delta \rangle$ if and only if the following holds for all $(q, \nu, q') \in R$:

1. $q \in F \implies q' \in F$, and
2. for all $\sigma \in \Sigma$ and all $(r, \nu') \in Q \times \mathcal{D}^{\mathbf{x}}$ such that $(q, \nu) \xrightarrow{\sigma}_A (r, \nu')$ there exists $r' \in Q$ such that $(q', \nu) \xrightarrow{\sigma}_A (r', \nu')$ and $(r, \nu', r') \in R$.

Observe that, while a classical simulation is a binary relation on states, a data simulation is a ternary relation that involves also a valuation of the variables. The following lemma shows that a data simulation preserves the residual languages of GRAs:

Lemma 9 *Given a GRA $A = \langle \Sigma, \mathcal{D}, \mathbf{x}, Q, \iota, F, \Delta \rangle$ and a data simulation $R \subseteq Q \times \mathcal{D}^{\mathbf{x}} \times Q$ for A, we have $\mathcal{L}_{(q,\nu)}(A) \subseteq \mathcal{L}_{(q',\nu)}(A)$ for any tuple $(q, \nu, q') \in R$.*

Proof Let $(\nu_0, \sigma_0), \ldots, (\nu_n, \diamond) \in \mathcal{L}_{(q,\nu)}(A)$ be a trace and $(q, \nu) = (q_0, \nu_0) \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_{n-1}} (q_n, \nu_n)$ be a run of A. By induction on $n \geqslant 0$, it is easy to find a run $(q', \nu) = (q'_0, \nu_0) \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_{n-1}} (q'_n, \nu_n)$ of A such that, for all $i \in [0,n]$, $(q_i, \nu_i, q'_i) \in R$ and moreover, $q_i \in F \implies q'_i \in F$. Thus, $(\nu_0, \sigma_0), \ldots, (\nu_n, \diamond) \in \mathcal{L}_{(q',\nu)}(A)$. $\qquad \square$

Let $A = \langle \Sigma, \mathcal{D}, \mathbf{x}, Q, \iota, F, \Delta \rangle$, where $Q = \{q_1, \ldots, q_k\}$ for some $k > 0$, be a GRA for the rest of this section. The data simulation algorithm (Algorithm 2) given in this section manipulates sets of valuations from $\mathcal{D}^{\mathbf{x}}$ that are definable by first-order formulae in $\text{Form}(\mathcal{D})$. A relation $R \subseteq Q \times \mathcal{D}^{\mathbf{x}} \times Q$ is said to be *definable* if and only if there exists a matrix $\Phi = [\phi_{ij}]_{i,j=1}^k$

Algorithm 2 Data Simulation Algorithm

---

input: A generic register automaton $A = \langle \Sigma, \mathcal{D}, \mathbf{x}, Q, \iota, F, \Delta \rangle$, where $Q = \{q_1, \dots, q_k\}$, and a constant $K > 0$.
output: A data simulation $R \subseteq Q \times \mathcal{D}^{\mathbf{x}} \times Q$ for A.
global vars $[Sim_{ij}]_{i,j=1}^{k}$, $[PrevSim_{ij}]_{i,j=1}^{k}$, $[Cnt_{ij}]_{i,j=1}^{k}$

1: **for** $i = 1, \dots, k$ **do**
2:     **for** $j = 1, \dots, k$ **do**
3:         $PrevSim_{ij} \leftarrow \top$
4:         $Cnt_{ij} \leftarrow K$
5:     **for** $j = 1, \dots, k$ **do**
6:     **if** $q_i \in F$ and $q_j \notin F$ **then**
7:         $Sim_{ij} \leftarrow \bot$
8:     **else**
9:         $Sim_{ij} \leftarrow \bigwedge_{\sigma \in \Sigma} \bigwedge_{q_\ell \in post_\sigma(q_i)} PreSim_\sigma(i,j,\ell,PrevSim)$
10: **while** $\exists \ell \in [1,k]$ such that $Sim_\ell \not\equiv PrevSim_\ell$ **do**
11:     $TempSim \leftarrow Sim$
12:     pick $\ell \in [1,k]$ such that $Sim_\ell \not\equiv PrevSim_\ell$
13:     **for** $\sigma \in \Sigma$ **do**
14:         **for** $q_i \in pre_\sigma(q_\ell)$ **do**
15:             **for** $j = 1, \dots, k$ **do**
16:                 $Sim_{ij} \leftarrow Sim_{ij} \wedge PreSim_\sigma(i,j,\ell,Sim)$
17:     **for all** $j = 1, \dots, k$ such that $Sim_{\ell j} \not\equiv PrevSim_{\ell j}$ **do**
18:         **if** $Cnt_{\ell j} = 0$ **then**
19:             $Sim_{\ell j} \leftarrow \bot$
20:         **else**
21:             $Cnt_{\ell j} \leftarrow Cnt_{\ell j} - 1$
22:     $PrevSim_\ell \leftarrow TempSim_\ell$
23: **return** Sim

---

of formulae $\phi_{ij}(\mathbf{x}) \in Form(\mathcal{D})$ such that $(q_i, \nu, q_j) \in R \iff \nu \models \phi_{ij}$. For $\ell \in [1,k]$, we denote by Φ_ℓ the ℓ-th row of the matrix Φ.

Algorithm 2 is a refinement algorithm which handles two matrices of formulae that define the relations $Sim, PrevSim \subseteq Q \times \mathcal{D}^{\mathbf{x}} \times Q$. Below, we shall use the same names to denote the relations and their matrix representations. Intuitively, $PrevSim$ is the previous candidate for simulation, whereas Sim is an entry-wise stronger relation that refines $PrevSim$. The refinement step is performed backwards wrt each transition rule $q_i \xrightarrow{\sigma, \phi} q_\ell$ of the automaton as follows. The tuple (q_i, ν, q_j) is added to the newly created relation Sim if $(q_i, \nu, q_j) \in PrevSim$ and there exist a valuation ν', a state $q_m \in Q$, and a formula ψ such that $(\nu, \nu') \models \phi$, $q_j \xrightarrow{\sigma, \psi} q_m$, $(q_\ell, \nu', q_m) \in PrevSim$, and $(\nu, \nu') \models \psi$. This update guarantees that, for every transition $(q_i, \nu) \xrightarrow{\sigma}_A (q_\ell, \nu')$ where $(q_i, \nu, q_j) \in Sim$, there exists a state q_m such that $(q_j, \nu) \xrightarrow{\sigma}_A (q_m, \nu')$ and $(q_\ell, \nu', q_m) \in PrevSim$. The algorithm stops when Sim and $PrevSim$ define the same relation. Moreover, this relation is guaranteed to be a data simulation.

To define the update, we use the following function, where $\sigma \in \Sigma$ is an input event, $i, j, \ell \in [1,k]$ are state indices such that $q_i \xrightarrow{\sigma, \phi} q_j \in \Delta$ is a transition rule and R is $k \times k$ matrix of formulae:

$$PreSim_\sigma(i,j,\ell,R) \equiv \forall \mathbf{x}' . \phi(\mathbf{x}, \mathbf{x}') \rightarrow \bigvee_{q_j \xrightarrow{\sigma, \psi} q_m} \psi(\mathbf{x}, \mathbf{x}') \wedge R_{\ell m}(\mathbf{x}') .$$

We also define the sets $post_\sigma(q) = \{q' \mid q \xrightarrow{\sigma, \phi} q' \in \Delta\}$ and $pre_\sigma(q) = \{q' \mid q' \xrightarrow{\sigma, \phi} q \in \Delta\}$. With this notation, Algorithm 2 describes the procedure that computes a data simulation for a given data automaton.

Initially, the matrix *PrevSim* is true everywhere (line 3). The current simulation candidate *Sim* is initialized to false for all $i,j \in [1,k]$ such that $q_i \in F$ and $q_j \notin F$ (line 7). Observe that, in this case, q_j cannot simulate q_i, by Definition 4 (1). Otherwise, we initialize Sim_{ij} to the strongest pre-simulation with respect to *PrevSim* (line 9). In the iterative loop (lines 10–22), the algorithm chooses a state q_ℓ for which the current simulation candidate Sim_ℓ is not equivalent to the previous one $PrevSim_\ell$ (line 13) and sharpens the set Sim_{ij} with respect to the transition rule $q_i \xrightarrow{\sigma,\phi} q_\ell$ for all input symbols $\sigma \in \Sigma$ and all peer states q_j, $j \in [1,k]$ (line 16). The following invariants are key to proving the correctness of Algorithm 2.

Lemma 10 *The following invariants hold each time Algorithm 2 reaches line 10:*
- *($SimInv_1$) for all $i,j \in [1,k]$, the entailment $Sim_{ij} \to PrevSim_{ij}$ is valid.*
- *($SimInv_2$) for all $\sigma \in \Sigma$, all $i,j,\ell \in [1,k]$ and all $v,v' \in \mathcal{D}^{\mathbf{x}}$, if $v \models Sim_{ij}$ and $(q_i,v) \xrightarrow{\sigma} (q_\ell,v')$ then there exists $m \in [1,k]$ such that $(q_j,v) \xrightarrow{\sigma} (q_m,v')$ and $v' \models PrevSim_{\ell m}$.*

Proof Let *Sim'* and *PrevSim'* denote the global matrices after one iteration of the loop on lines 10–22.

($SimInv_1$) When line 10 is reached for the first time, $PrevSim_{ij} = \top$ for all $i,j \in [1,k]$, thus $SimInv_1$ holds initially. Since *Sim* is modified on lines 16 or 19 only, we have $Sim'_{ij} \to Sim_{ij}$ for all $i,j \in [1,k]$. Moreover, for each $i,j \in [1,k]$ either (i) $PrevSim'_{ij} = TempSim_{ij} = Sim_{ij}$ (line 22) and $Sim'_{ij} \to Sim_{ij} \to PrevSim'_{ij}$ holds, or (ii) $PrevSim'_{ij} = PrevSim_{ij}$ (no update) and $Sim'_{ij} \to Sim_{ij} \to PrevSim_{ij} \to PrevSim'_{ij}$ holds, by the inductive hypothesis.

($SimInv_2$) We show that this invariant holds the first time the control reaches line 10. Let $\sigma \in \Sigma$, $i,j,\ell \in [1,k]$ and $v,v' \in \mathcal{D}^{\mathbf{x}}$ such that $v \models Sim_{ij}$ and $(q_i,v) \xrightarrow{\sigma} (q_\ell,v')$. Since $v \models Sim_{ij}$ (thus $Sim_{ij} \neq \bot$) and $q_\ell \in post_\sigma(q_i)$, we have that $v \models PreSim_\sigma(i,j,\ell,PrevSim)$ where $q_i \xrightarrow{\sigma,\phi} q_\ell \in \Delta$. Since $(q_i,v) \xrightarrow{\sigma} (q_\ell,v')$, we obtain that $(v,v') \models \phi(\mathbf{x},\mathbf{x}')$, and, consequently, $(v,v') \models \psi(\mathbf{x},\mathbf{x}') \wedge PrevSim_{\ell m}(\mathbf{x}')$ for some $m \in [1,k]$ such that $q_j \xrightarrow{\sigma,\psi} q_m \in \Delta$. Hence, $SimInv_2$ holds when the control first reaches line 10.

For the induction step, let us assume that $SimInv_2$ holds on line 10, and we prove that it also holds after executing line 22. Let $\sigma \in \Sigma$, $i,j,\ell \in [1,k]$ and $v,v' \in \mathcal{D}^{\mathbf{x}}$ such that $v \models Sim'_{ij}$ and $(q_i,v) \xrightarrow{\sigma} (q_\ell,v')$. We distinguish two cases:

1. If $Sim_\ell \not\equiv PrevSim_\ell$ on line 10 since $q_i \in pre_\sigma(q_\ell)$, then Sim'_{ij} was updated on line 16. Since $v \models Sim'_{ij}$, we obtain $v \models PreSim_\sigma(q_i,q_j,q_\ell,Sim)$. Moreover, $PrevSim'_\ell$ is updated to $TempSim_\ell \equiv Sim_\ell$ on line 22, hence $v \models PreSim_\sigma(q_i,q_j,q_\ell,PrevSim')$ as well. Since $(q_i,v) \xrightarrow{\sigma} (q_\ell,v')$, we obtain that $(v,v') \models \psi(\mathbf{x},\mathbf{x}') \wedge PrevSim'_{\ell m}(\mathbf{x}')$ for some $m \in [1,k]$ such that $q_j \xrightarrow{\sigma,\psi} q_m \in \Delta$, thus $(v,v') \models \psi(\mathbf{x},\mathbf{x}')$ and $v' \models PrevSim'_{\ell m}$. Thus $SimInv_2$ holds for *Sim'* and *PrevSim'*.

2. Otherwise $Sim_\ell \equiv PrevSim_\ell$ on line 10. Moreover, $PrevSim'_\ell \equiv PrevSim_\ell$ because the update on line 22 is skipped, and, for all $q_i \in pre_\sigma(q_\ell)$ and all $j \in [1,k]$, we have $Sim'_{ij} \equiv Sim_{ij}$. Then, by the induction hypothesis, $SimInv_2$ holds for *Sim'* and *PrevSim'* because it holds for *Sim* and *PrevSim*.

$\square$

The algorithm iterates the loop on lines (10–22) until *Sim* and *PrevSim* define the same relation. Since, in general, the data constraints Sim_{ij} obtained from different iteration steps might form an infinitely decreasing sequence, we use the matrix *Cnt* of integer counters,

initially set to some input value $K > 0$ (line 4).[12] Observe that each entry Cnt_{ij} decreases every time $Sim_{ij} \not\equiv PrevSim_{ij}$ (line 21). When the counter Cnt_{ij} reaches zero, we set Sim_{ij} to false (line 19), which guarantees that $Sim_{ij} \equiv PrevSim_{ij}$ always in the future. Since the number of entries in the counter matrix is finite, the algorithm is guaranteed to terminate. The following theorem summarizes the main result of this section.

Theorem 2 *Algorithm 2 terminates on any GRA $A = \langle \Sigma, \mathcal{D}, \mathbf{x}, Q, \iota, F, \Delta \rangle$, and its output is a data simulation $R \subseteq Q \times \mathcal{D}^{\mathbf{x}} \times Q$ for A.*

Proof Let Sim^n and $PrevSim^n$ denote the matrices Sim and $PrevSim$ at the n-th iteration of the loop on lines 10–22, for $n \geqslant 0$. Algorithm 2 terminates whenever $Sim_{ij}^n \equiv PrevSim_{ij}^n$ for all $i, j \in [1, k]$ (line 10). Suppose, by contradiction, that this never happens, thus there exist $i, j \in [1, k]$ such that $Sim_{ij}^n \not\equiv PrevSim_{ij}^n$ for all $n \geqslant 0$. Then $Cnt_{ij}^K = 0$ (line 21) and $Sim_{ij}^{K+1} = PrevSim_{ij}^{K+2} = \bot$ (lines 19 and 22). Since $Sim_{ij}^n \rightarrow PrevSim_{ij}^n$, by Lemma 10 ($SimInv_1$), we obtain that $Sim_{ij}^{K+2} = PrevSim_{ij}^{K+2}$, a contradiction.

To prove that the output of Algorithm 2 is a data simulation for A, we use Lemma 10 ($SimInv_2$) and the fact that, upon termination, we have $Sim_{ij} \equiv PrevSim_{ij}$, for all $i, j \in [1, k]$. □

5.2 Simulation and Subsumption

Finally, we explain how a data simulation relation computed by Algorithm 2 can be used to optimize the trace inclusion semi-algorithm. Let $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ be a GRAN where $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$ for all $i \in [1, N]$, and let $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ be an observer GRA such that $\mathbf{x}_B \subseteq \bigcup_{i=1}^{N} \mathbf{x}_i$.

The main problem for using data simulations to enhance the convergence of our trace inclusion semi-algorithm is related to the fact that simulation relations are, in general, not compositional wrt the interleaving semantics of the network. In other words, if we have N data simulations $R_i \subseteq Q_i \times \mathcal{D}^{\mathbf{x}_i} \times Q_i$ for $i \in [1, N]$, then their cross-product $R \subseteq Q_{\mathcal{A}} \times \mathcal{D}^{\mathbf{x}_{\mathcal{A}}} \times Q_{\mathcal{A}}$ defined as:

$$\forall q_1, r_1 \in Q_1 \ldots \forall q_N, r_N \in Q_N \forall \mathbf{v} \in \mathcal{D}^{\mathbf{x}_{\mathcal{A}}} : (\langle q_1, \ldots, q_N \rangle, \mathbf{v}, \langle r_1, \ldots, r_N \rangle) \in R \iff (q_i, \mathbf{v} \downarrow_{\mathbf{x}_i}, r_i) \in R_i$$

is not necessarily a simulation on the network expansion $\mathcal{A}^e$. The reason for this can be seen for $N = 2$. Let $\sigma_1, \sigma_2 \in \Sigma_{\mathcal{A}}$ such that $\sigma_1 \notin \Sigma_2$ and $\sigma_2 \notin \Sigma_1$. The execution of $\mathcal{A}^e$ on the sequence of input symbols $\sigma_1 \sigma_2$ is $(\langle q_1, q_2 \rangle, \mathbf{v}) \xrightarrow{\sigma_1} (\langle q_1', q_2 \rangle, \mathbf{v}') \xrightarrow{\sigma_2} (\langle q_1', q_2' \rangle, \mathbf{v}'')$. Suppose that $(q_i, \mathbf{v} \downarrow_{\mathbf{x}_i}, r_i) \in R_i$, for all $i = 1, 2$. Then there exists $r_1' \in Q_1$ such that $(\langle r_1, r_2 \rangle, \mathbf{v}) \xrightarrow{\sigma_1} (\langle r_1', r_2 \rangle, \mathbf{v}')$ and $(q_1', \mathbf{v}' \downarrow_{\mathbf{x}_1}, r_1') \in R_1$. In order to use the simulation and build the continuation $(\langle r_1', r_2 \rangle, \mathbf{v}') \xrightarrow{\sigma_2} (\langle r_1', r_2' \rangle, \mathbf{v}'')$, we would need that $(q_2, \mathbf{v}' \downarrow_{\mathbf{x}_2}, r_2) \in R_2$, which is not necessarily ensured by the hypothesis $(q_2, \mathbf{v} \downarrow_{\mathbf{x}_2}, r_2) \in R_2$.

We propose a partial solution to this problem, based on a restriction concerning the distribution of the network variables $\mathbf{x}_{\mathcal{A}} = \bigcup_{i=1}^{N} \mathbf{x}_i$ over the components $A_1, \ldots, A_N$: for each $i \in [1, N]$, we have $\mathbf{x}_i = \mathbf{x}^g \cup \mathbf{x}_i^\ell$ where $\mathbf{x}^g$ is a set of *global* variables and $\mathbf{x}_i^\ell$ are the *local* variables of A_i. In other words, the only variables shared between more than one component are the global variables $\mathbf{x}^g$, which, moreover, are visible to all components.[13] Then the problem

---

[12] Taking a bigger K leads to a more precise Sim_{ij}, but, on the other hand, it can significantly increase the computation time.

[13] Many realistic systems comply with this restriction, take, for instance, shared-memory multithreading in Java.

can be bypassed if none of the simulation relations $R_i \subseteq Q_i \times \mathcal{D}^{\mathbf{x}_i} \times Q_i$ may constrain the global variables:

Assumption 3 *For each $i \in [1, N]$ and each $(q_i, \nu, r_i) \in R_i$, we also have $(q_i, \nu', r_i) \in R_i$ for each $\nu' \in \mathcal{D}^{\mathbf{x}_i}$ such that $\nu \downarrow_{\mathbf{x}_i^\ell} = \nu' \downarrow_{\mathbf{x}_i^\ell}$.*

Under this assumption, we use pre-computed data simulations $R_i \subseteq Q_i \times \mathcal{D}^{\mathbf{x}^g} \times Q_i$ and $R_B \subseteq Q_B \times \mathcal{D}^{\mathbf{x}_B} \times Q_B$ to generalize the basic subsumption relation between product states (defined by Lemma 4), which may speed up the convergence of Algorithm 1.

Lemma 11 *Under Assumption 3, the relation defined as*

$$(\langle q_1, \ldots, q_N \rangle, P, \Phi) \sqsubseteq_{sim} (\langle r_1, \ldots, r_N \rangle, S, \Psi)$$
$$\Longleftrightarrow$$
$$\forall i \in [1, N] \; \forall \nu \in \mathcal{D}^{\mathbf{x}_\mathcal{A}} : \nu \models \Phi \Longrightarrow \nu \models \Psi \; and \; \begin{cases} (q_i, \nu \downarrow_{\mathbf{x}_i}, r_i) \in R_i & (1) \\ \forall s \in S \; \exists p \in P \; : \; (s, \nu \downarrow_{\mathbf{x}_B}, p) \in R_B & (2) \end{cases}$$

is a subsumption relation.

Proof Let $s = (\langle q_1, \ldots, q_N \rangle, P, \Phi)$ and $t = (\langle r_1, \ldots, r_N \rangle, S, \Psi)$ be two product states such that $s \sqsubseteq_{sim} t$. According to Definition 3, we need to prove that $\mathcal{L}_s(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_t(\mathcal{A}^e \times \overline{B})$. For that, it is sufficient to prove that, for each $\nu \in \mathcal{D}^{\mathbf{x}_\mathcal{A}}$ such that $\nu \models \Phi$, the following two points hold:
1. $\mathcal{L}_{(\langle q_1, \ldots, q_N \rangle, \nu)}(\mathcal{A}^e) \subseteq \mathcal{L}_{(\langle r_1, \ldots, r_N \rangle, \nu)}(\mathcal{A}^e)$, and
2. for all $p \in S$ there exists $q \in P$ such that $\mathcal{L}_{(p, \nu \downarrow_{\mathbf{x}_B})}(B) \subseteq \mathcal{L}_{(q, \nu \downarrow_{\mathbf{x}_B})}(B)$.

Indeed, assuming that the above statements hold, we have

$$\begin{aligned} \mathcal{L}_s(\mathcal{A}^e \times \overline{B}) &= \bigcup\nolimits_{\nu \models \Phi} \left(\mathcal{L}_{(\langle q_1, \ldots, q_N \rangle, \nu)}(\mathcal{A}^e) \cap \bigcap\nolimits_{q \in P} \mathcal{L}_{(q, \nu \downarrow_{\mathbf{x}_B})}(\overline{B}) \right) \\ &\subseteq \bigcup\nolimits_{\nu \models \Phi} \left(\mathcal{L}_{(\langle r_1, \ldots, r_N \rangle, \nu)}(\mathcal{A}^e) \cap \bigcap\nolimits_{r \in S} \mathcal{L}_{(r, \nu \downarrow_{\mathbf{x}_B})}(\overline{B}) \right) \\ &\subseteq \bigcup\nolimits_{\nu \models \Psi} \left(\mathcal{L}_{(\langle r_1, \ldots, r_N \rangle, \nu)}(\mathcal{A}^e) \cap \bigcap\nolimits_{r \in S} \mathcal{L}_{(r, \nu \downarrow_{\mathbf{x}_B})}(\overline{B}) \right) \\ &= \mathcal{L}_t(\mathcal{A}^e \times \overline{B}), \end{aligned}$$

and we are done. Moreover, the second point above is a direct consequence of the second point of the definition of $\sqsubseteq_{sim}$ and Lemma 9. We are left with proving the first point.

To prove the first point, assume that we are given configurations $(\langle q_0^1, \ldots, q_0^N \rangle, \nu_0)$ and $(\langle r_0^1, \ldots, r_0^N \rangle, \nu_0)$ of $\mathcal{A}^e$ such that $\forall i \in [1, N] : (q_0^i, \nu_0 \downarrow_{\mathbf{x}_i}, r_0^i) \in R_i$. We show that if there is a run $(\langle q_0^1, \ldots, q_0^N \rangle, \nu_0) \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_{n-1}} (\langle q_n^1, \ldots, q_n^N \rangle, \nu_n)$ for any $n \geq 0$, then there is some run $(\langle r_0^1, \ldots, r_0^N \rangle, \nu_0) \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_{n-1}} (\langle r_n^1, \ldots, r_n^N \rangle, \nu_n)$ where $\forall i \in [1, N] : \forall j \in [0, n] : (q_j^i, \nu_j \downarrow_{\mathbf{x}_i}, r_j^i) \in R_i \wedge (q_j^i \in F_i \Longrightarrow r_j^i \in F_i)$, by induction on the length n of the run. The base case for $n = 0$ follows trivially from the assumption $\forall i \in [1, N] : (q_0^i, \nu_0 \downarrow_{\mathbf{x}_i}, r_0^i) \in R_i$ and from the first point of the definition of data simulations (Def. 4). Now, assuming that the property holds for runs of length n, we show that it holds for runs of length $n + 1$ too. Take a run $(\langle q_0^1, \ldots, q_0^N \rangle, \nu_0) \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_{n-1}} (\langle q_n^1, \ldots, q_n^N \rangle, \nu_n) \xrightarrow{\sigma_n} (\langle q_{n+1}^1, \ldots, q_{n+1}^N \rangle, \nu_{n+1})$. Further, take a configuration $(\langle r_0^1, \ldots, r_0^N \rangle, \nu_0)$ such that $\forall i \in [1, N] : (q_0^i, \nu_0 \downarrow_{\mathbf{x}_i}, r_0^i) \in R_i$. From the induction hypothesis, we immediately get that there exists a run $(\langle r_0^1, \ldots, r_0^N \rangle, \nu_0) \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_{n-1}} (\langle r_n^1, \ldots, r_n^N \rangle, \nu_n)$ where $\forall i \in [1, N] : \forall j \in [0, n] : (q_j^i, \nu_j \downarrow_{\mathbf{x}_i}, r_j^i) \in R_i \wedge (q_j^i \in F_i \Longrightarrow r_j^i \in F_i)$. Next, let $I \subseteq [0, N]$ be the set of indices of the GRAs that make a move during the step $(\langle q_n^1, \ldots, q_n^N \rangle, \nu_n) \xrightarrow{\sigma_n} (\langle q_{n+1}^1, \ldots, q_{n+1}^N \rangle, \nu_{n+1})$. For any $i \in I$, from $(q_n^i, \nu_n \downarrow_{\mathbf{x}_i}, r_n^i) \in R_i$ and $(q_n^i, \nu_n \downarrow_{\mathbf{x}_i}) \xrightarrow{\sigma_n} (q_{n+1}^i, \nu_{n+1} \downarrow_{\mathbf{x}_i})$, we get there there is some r_{n+1}^i such that $(r_n^i, \nu_n \downarrow_{\mathbf{x}_i}) \xrightarrow{\sigma_n} (r_{n+1}^i, \nu_{n+1} \downarrow_{\mathbf{x}_i})$, $(q_{n+1}^i, \nu_{n+1} \downarrow_{\mathbf{x}_i}, r_{n+1}^i) \in R_i$, and $q_{n+1}^i \in F_i \to r_{n+1}^i \in F_i$. Moreover, for any

Table 1 Experiments with single-component networks.

Example	A ($|Q|/|\Delta|$)	B ($|Q|/|\Delta|$)	Vars.	Res.	Time
Arrays shift	3/3	3/4	5	ok	$< 0.1s$
Array rotation 1	4/5	4/5	7	ok	$0.1s$
Array rotation 2	8/21	6/24	11	ok	$34s$
Array split	20/103	6/26	14	ok	$4m32s$
HW counter 1	2/3	1/2	2	ok	$0.2s$
HW counter 2	6/12	1/2	2	ok	$0.4s$
Synchr. LIFO	4/34	2/15	4	ok	$2.5s$
ABP-error	14/20	2/6	14	cex	$2s$
ABP-correct	14/20	2/6	14	ok	$3s$

$i \in [1, N] \setminus I$, the fact that $\nu_n \downarrow_{\mathbf{x}_i^l} = \nu_{n+1} \downarrow_{\mathbf{x}_i^l}$, $(q_n^i, \nu_n \downarrow_{\mathbf{x}_i}, r_n^i) \in R_i$, $q_{n+1}^i = q_n^i$, $r_{n+1}^i = r_n^i$, and Assumption 3, give us $(q_{n+1}^i, \nu_{n+1} \downarrow_{\mathbf{x}_i}, r_{n+1}^i) \in R_i$, and, consequently, $q_{n+1}^i \in F_i \rightarrow r_{n+1}^i \in F_i$ too. $\qquad \square$

6 Experimental Results

We have implemented both Algorithm 1 (trace inclusion) and Algorithm 2 (data simulations) in a prototype tool INCLUDER[14] using the MATHSAT SMT solver [9] for answering the satisfiability queries and computing the interpolants. The results of the experiments with trace inclusion are given in Tables 1 and 2. The results of experiments combining trace inclusion and simulations are given in Table 3. The results were obtained on an Intel i7-4770 CPU @ 3.40GHz machine with 32GB RAM.

6.1 Trace Inclusion

Table 1 contains experiments where the network $\mathcal{A}$ consists of a single component. We applied the tool on several verification conditions generated from imperative programs with arrays [8] (Array shift, Array rotation 1+2, Array split) available online [29]. Then, we applied it on models of hardware circuits (HW Counter 1+2, Synchronous LIFO) [31]. Finally, we checked two versions (correct and faulty) of the timed Alternating Bit Protocol [34].

Table 2 provides a list of experiments where the network $\mathcal{A}$ has $N > 1$ components. First, we have the example of Fig. 1 (Running). Next, we have several examples of real-time verification problems [32]: a controller of a railroad crossing [21] (Train) with T trains, the Fischer Mutual Exclusion protocol with deadlines Δ and Γ (Fischer), and a hardware communication circuit with K stages, composed of timed NOR gates (Stari). Third, we have modeled a Producer-Consumer example [14] with a fixed buffer size B. Fourth, we have experimented with several models of parallel programs that manipulate arrays (Array init, Array copy, Array join) with window size Δ.

For the time being, our implementation is a proof-of-concept prototype that leaves plenty of room for optimization (e.g., caching of intermediate computation results) likely to improve the performance on more complicated examples. Despite that, we found the results from Tables 1 and 2 rather encouraging.

---

[14] http://www.fit.vutbr.cz/research/groups/verifit/tools/includer/

Table 2 Experiments with multiple-component networks (e.g., $2 \times 2/2 + 2 \times 3/3$ in column $\mathcal{A}$ means that $\mathcal{A}$ is a network with 4 components that includes 2 GRAs with 2 states and 2 rules and 2 GRAs with 3 states and 3 rules).

Example	N	$\mathcal{A}$ ($|Q|/|\Delta|$)	B ($|Q|/|\Delta|$)	Vars.	Res.	Time
Running	2	$2 \times 2/2$	3/4	3	ok	0.2s
Running	10	$10 \times 2/2$	11/20	3	ok	25s
Train ($T = 5$)	7	$5 \times 3/3 + 4/4 + 4/4$	2/38	1	ok	4s
Train ($T = 10$)	12	$10 \times 3/3 + 4/4 + 4/4$	2/68	1	ok	29s
Train ($T = 20$)	22	$20 \times 3/3 + 4/4 + 4/4$	2/128	1	ok	6m26s
Fischer ($\Delta = 1, \Gamma = 2$)	2	$2 \times 5/6$	1/10	4	ok	8s
Fischer ($\Delta = 1, \Gamma = 2$)	3	$3 \times 5/6$	1/15	4	ok	2m48s
Fischer ($\Delta = 2, \Gamma = 1$)	2	$2 \times 5/6$	1/10	4	cex	3s
Fischer ($\Delta = 2, \Gamma = 1$)	3	$3 \times 5/6$	1/15	4	cex	32s
Stari ($K = 1$)	5	$4/5 + 2/4 + 5/7 + 5/7 + 5/7$	3/6	3	ok	0.5s
Stari ($K = 2$)	8	$4/5 + 2/4 + 2 \times 5/7 + 2 \times 5/7 + 2 \times 5/7$	3/6	3	ok	0.5s
Prod-Cons ($B = 3$)	2	$4/4 + 4/4$	2/7	2	ok	10s
Prod-Cons ($B = 6$)	2	$4/4 + 4/4$	2/7	2	ok	2m32s
Array init ($\Delta = 2$)	5	$5 \times 2/2$	2/6	2	ok	3s
Array init ($\Delta = 2$)	15	$15 \times 2/2$	2/16	2	ok	3m15s
Array copy ($\Delta = 20$)	20	$20 \times 2/2$	2/21	3	ok	0.3s
Array copy ($\Delta = 20$)	150	$150 \times 2/2$	2/151	3	ok	43s
Array join ($\Delta = 10$)	4	$2 \times 2/2 + 2 \times 3/3$	2/3	2	ok	6s
Array join ($\Delta = 10$)	6	$3 \times 2/2 + 3 \times 3/3$	2/4	2	ok	23s
Array join ($\Delta = 20$)	6	$3 \times 2/2 + 3 \times 3/3$	2/4	2	ok	1m9s

6.2 Combination of Trace Inclusion and Simulations

Unlike the computation of the most general simulation on a finite-alphabet automaton, which is possible in polynomial time [**?**], computing the weakest data simulation on a GRA is, in general, impossible due to the fact that the data constraints cannot be represented in a decidable logical domain, such as linear integer arithmetic. For this reason, our algorithm (Algorithm 2) is sound but incomplete, returning a possibly stronger simulation, in which the data constraint associated with certain pairs of states is set of $\perp$. Such simulations can be computed in reasonable time, but they could be of limited use in speeding up the antichain-based trace inclusion check.

Our implementation tries to achieve a balance between these opponent goals, as shown by the results in Table 3. We apply a timeout on each single call of the *PreSim* function in Algorithm 2. Moreover, we may also limit the size of the resulting formula[15] for a single call of the *PreSim* function. If the timeout or the size limit is exceeded, the result of *PreSim* will be safely underapproximated to $\perp$ (i.e., no simulation).

The use of simulation-based subsumption has an impact on running times of the trace inclusion in the examples, where the system contains a nontrivial simulation relation, which could be discovered by Algorithm 2 at most $K = 2$ iterations on each pair of states and, moreover, some product states in the antichain tree are compatible with this simulation relation. This is clearly visible in the Fischer 2-serial and Fischer 2-branching models where the synchronization based on the Fischer protocol is used in non-minimalistic scenarios, i.e., scenarios not restricted to a single critical section. In particular, Fischer 2-serial is an abstract model of a system where a process uses the Fischer protocol to access a critical section twice in a row. Fischer 2-branching is an abstraction of a system where a single process contains two branches and each of these branches accesses a critical section using the Fischer's protocol. The parameters Δ and Γ are parameters of the Fischer's protocol, and N is the number

---

[15] The size of a formula is measured in the number of nodes of its MathSAT graph-based representation.

Table 3 Experiments with simulations. $Time_{inclusion}^{nosim}$ represents the time of checking trace inclusion without using simulation, $Time_{inclusion}^{sim}$ represents the time of checking trace inclusion while using simulation-based subsumption, and $Time_{sim}$ represents the time needed to compute data simulations. K is the parameter of Algorithm 2, *Max size* is a limit on the size of the result of the *PreSim* function, and *TO* is a timeout on a single call of the *PreSim* function.

Example	$Time_{inclusion}^{nosim}$	$Time_{inclusion}^{sim}$	$Time_{sim}$	K	*Max size*	*TO*
Array rotation 2	27s	24s	38s	2	15	2s
Array rotation 2	27s	24s	24s	2	15	1s
Array Split	3m23s	2m37s	5m52s	2	15	2s
Array Split	3m23s	2m37s	4m1s	2	15	1s
Fischer 2-serial ($\Delta = 1, \Gamma = 2, N = 3$)	17m43s	3m55s	2s	2	∞	∞
Fischer 2-serial ($\Delta = 1, \Gamma = 2, N = 2$)	27s	9s	2s	2	∞	∞
Fischer 2-serial ($\Delta = 2, \Gamma = 1, N = 3$) (cex)	50s	1m	2s	2	∞	∞
Fischer 2-serial ($\Delta = 2, \Gamma = 1, N = 2$) (cex)	3s	2s	2s	2	∞	∞
Fischer 2-branch ($\Delta = 1, \Gamma = 2, N = 3$)	25m57s	4m28s	2s	2	∞	∞
Fischer 2-branch ($\Delta = 1, \Gamma = 2, N = 2$)	24s	11s	2s	2	∞	∞
Fischer 2-branch ($\Delta = 2, \Gamma = 1, N = 3$) (cex)	1m49s	1m8s	2s	2	∞	∞
Fischer 2-branch ($\Delta = 2, \Gamma = 1, N = 2$) (cex)	10s	3s	2s	2	∞	∞

of parallel processes. Note that if the system contains a counterexample (e.g. Fischer 2-serial with parameters $\Delta = 2, \Gamma = 1, N = 3$), the simulation-based subsumption may also increase the running time. The reason is that the computation is stopped when an accepting product state is discovered and the rest of the antichain tree is not constructed. The simulation-based subsumption makes the whole antichain tree smaller (in terms of nodes), but the shortest counterexample path may be subsumed by a longer one resulting to a postpone of a counterexample discovery.

Also note that we managed to compute nontrivial simulations[16] for all the examples from Tables 1 and 2. However, in most of them, the use of the simulation has no impact on the time of checking the trace inclusion. The main reason is that most of the protocols are modeled by automata where very limited data simulations exist between pairs of states (i.e., the data constraints under which the simulation holds are quite strong), and product states in the antichain tree are incompatible with these simulations (cf. Points 1 and 2 of Lemma 11).

7 Conclusions

We have presented an interpolation-based abstraction refinement method for trace inclusion between a network of generic register automata and an observer where the variables used by the observer are a subset of those used by the network. The procedure builds on a new determinization result for GRAs and combines in a novel way predicate abstraction and interpolation with antichain-based inclusion checking. The efficiency of the basic method can be further enhanced by data simulations. The procedure has been successfully applied to several examples, including verification problems for array programs, real-time systems, and hardware designs.

For the future, it is interesting to extend the method to data tree automata and apply it to logics for heaps with data. Also, we foresee an extension of the method to handle infinite traces. Finally, it is also an open problem how to handle the case when the observer is allowed to have local variables.

---

[16] A simulation R is trivial iff $\forall x, y \in Q : x \neq y \rightarrow (x, \perp, y) \in R$.

References

1. Abdulla, P., Chen, Y.F., Holik, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Proc. of TACAS'10, *LNCS*, vol. 6015, pp. 158–174. Springer (2010)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
3. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: Fast acceleration of symbolic transition systems. In: Proc. of CAV'03, *LNCS*, vol. 2725. Springer (2003)
4. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Proc. of CAV'13, *LNCS*, vol. 8044. Springer (2013)
5. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn Clause Solvers for Program Verification, pp. 24–51. Springer International Publishing, Cham (2015)
6. Bojańczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Trans. Comput. Logic **12**(4), 27:1–27:26 (2011)
7. Bonchi, F., Pous, D.: Checking nfa equivalence with bisimulations up to congruence. In: Proc. of POPL'13. ACM (2013)
8. Bozga, M., Habermehl, P., Iosif, R., Konecný, F., Vojnar, T.: Automatic verification of integer array programs. In: Proc. of CAV'09, *LNCS*, vol. 5643, pp. 157–172 (2009)
9. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Proc. of TACAS, *LNCS*, vol. 7795 (2013)
10. Cook, B., Khlaaf, H., Piterman, N.: On automation of ctl* verification for infinite-state systems. In: Proc. of CAV'15, *LNCS*, vol. 9206. Springer (2015)
11. Craig, W.: Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
12. D'Antoni, L., Alur, R.: Symbolic visibly pushdown automata. In: Proc. of CAV'14, *LNCS*, vol. 8559. Springer (2014)
13. Decker, N., Habermehl, P., Leucker, M., Thoma, D.: Ordered navigation on multi-attributed data words. In: Proc. of CONCUR'14, *LNCS*, vol. 8704, pp. 497–511 (2014)
14. Dhar, A.: Algorithms for model-checking flat counter systems. Ph.D. thesis, Univ. Paris 7 (2014)
15. Fribourg, L.: A closed-form evaluation for extended timed automata. Tech. rep., CNRS et Ecole Normale Supérieure de Cachan (1998)
16. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, pp. 405–416 (2012)
17. Habermehl, P., Iosif, R., Vojnar, T.: A logic of singly indexed arrays. In: Proc. of LPAR'08, *LNCS*, vol. 5330, pp. 558–573 (2008)
18. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Proc. of FOS-SACS'08, *LNCS*, vol. 4962, pp. 474–489 (2008)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. of POPL'02. ACM (2002)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In: Proc. of 10th SPIN Workshop, *LNCS*, vol. 2648 (2003)
21. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation **111**, 394–406 (1992)
22. Iosif, R., Rogalewicz, A., Vojnar, T.: Abstraction refinement and antichains for trace inclusion of infinite state systems. In: Proc. of TACAS'16, *LNCS*, vol. 9636, pp. 71–89. Springer (2016)
23. Iosif, R., Xu, X.: Abstraction refinement for emptiness checking of alternating data automata. In: Proc. TACAS'18, *LNCS*, vol. 10806, pp. 93–111. Springer (2018)
24. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science **134**(2), 329 – 363 (1994). DOI https://doi.org/10.1016/0304-3975(94)90242-9
25. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. of CAV'06, *LNCS*, vol. 4144. Springer (2006)
26. McMillan, K.L.: Interpolants from z3 proofs. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, pp. 19–27. FMCAD Inc (2011)
27. Milner, R.: An algebraic definition of simulation between programs. In: Proc. of IJCAI'71. Morgan Kaufmann Publishers Inc. (1971)
28. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall (1967)
29. Numerical Transition Systems Repository. http://nts.imag.fr/index.php/Flata (2012)
30. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: Closing a decidability gap. In: Proc of LICS'04. IEEE Computer Society (2004)
31. Smrcka, A., Vojnar, T.: Verifying parametrised hardware designs via counter automata. In: HVC'07, pp. 51–68 (2007)

32. Tripakis, S.: The analysis of timed systems in practice. Ph.D. thesis, Univ. Joseph Fourier, Grenoble (December 1998)
33. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.: Antichains: A new algorithm for checking universality of finite automata. In: Proc. of CAV'06, *LNCS*, vol. 4144. Springer (2006)
34. Zbrzezny, A., Polrola, A.: Sat-based reachability checking for timed automata with discrete data. Fundamenta Informaticae **79**, 1–15 (2007)

A Alternative Notions of the Product State

Below, we briefly discuss two alternative notions of product states that we originally considered but dropped them since we were not able to build a sound antichain construction on them.

The first option we considered was to link predicates with the *individual* states involved in a product state. In that case, the predicate map linked particular states of automata $\mathcal{A}^e$ and B to sets of formulas as follows: $\Pi_{ind} : Q_{\mathcal{A}^e} \cup Q_B \to 2^{\mathrm{Form}(\mathcal{D})}$. The product state was then defined as $s_{ind} = (\langle \mathbf{q}, \Phi_q \rangle, P)$ with $\mathbf{q}$ being a state of the automaton $\mathcal{A}^e$, $\Phi_q \subseteq \Pi_{ind}(\mathbf{q})$, and $P \subseteq \{\langle r, \Phi_r \rangle \mid r \in Q_B \text{ and } \Phi_r \subseteq \Pi_{ind}(r)\}$. The semantics of the product state $s_{ind} = (\langle \mathbf{q}, \Phi_q \rangle, P)$ was that whenever the automaton $\mathcal{A}^e$ is in the state $\mathbf{q}$ with a valuation $\nu \models \Phi_q$ of the variables, then the automaton B can be in any state r such that $\langle r, \Phi_r \rangle \in P$ and $\nu \models \Phi_r$. A product state $s_{ind} = (\langle \mathbf{q}, \Phi_q \rangle, P)$ was considered accepting iff $\mathbf{q} \in F_{\mathcal{A}^e}$ and there existed $\nu \models \Phi_q$ such that $\nu \models \bigvee \{\Phi_r \mid \langle r, \Phi_r \rangle \in P \wedge r \in F_B\}$. That implied existence of a trace accepted by $\mathcal{A}^e$ at the state $\mathbf{q}$ with the final valuation ν, not covered by the automaton B. A problem with this product construction is that it cannot be used for soundly deciding the inclusion problem as shown in the following example: Take the product state $s_1 = (\langle q_1, x \in \{1, 2\} \rangle, \{\langle r_1, x = 1 \rangle, \langle r_2, x = 2 \rangle\})$ obtained for an automaton $\mathcal{A}^e$ with the rule $q_1 \xrightarrow{\sigma, x' = x+1} q_2$ and an automaton B with rules $r_1 \xrightarrow{\sigma, x' > x} r_3$ and $r_2 \xrightarrow{\sigma, x' = x+1 \wedge x > 10} r_3$. Moreover, let q_2 be final in $\mathcal{A}^e$ and r_3 be final in B. When one computes the post of s_1, one gets $s_2 = (\langle q_2, x \in \{2, 3\} \rangle, \{\langle r_3, x > 1 \rangle\})$, which is not accepting, because all configurations of $\mathcal{A}^e$ are covered by configurations of B. However, the automaton $\mathcal{A}^e$ can do a step $(q_1, x = 2) \xrightarrow{\sigma, x' = x+1} (q_2, x = 3)$, which cannot be followed by B. Hence, an antichain construction based on this notion of product states could hide a real counterexample and provide an unsound answer.

In order to avoid the unsoundness of the above solution, we attempted to use predicates representing *relations* between successive values of variables within a step leading to a given product state. In this case, the predicate map was defined as $\Pi_{rel} : Q_{\mathcal{A}^e} \cup Q_B \to 2^{\mathrm{Form}(\mathcal{D})} \times 2^{\mathrm{Form}(\mathcal{D})}$. The product state was then defined as $s_{rel} = (\langle \mathbf{q}, \Phi_q \rangle, P)$ with $\mathbf{q}$ being a state of the automaton $\mathcal{A}^e$, $\Phi_q \subseteq \Pi_{rel}(\mathbf{q})$, and $P \subseteq \{\langle r, \Phi_r \rangle \mid r \in Q_B \text{ and } \Phi_r \subseteq \Pi_{rel}(r)\}$. The semantics of the product state $s_{rel} = (\langle \mathbf{q}, \Phi_q \rangle, P)$ was that whenever the last step of $\mathcal{A}^e$ was $(\_, \nu) \to (\mathbf{q}, \nu')$ such that $(\nu, \nu') \models \Phi_r$, then the last step of B could have been $(\_, \nu) \to (r, \nu')$ where $\langle r, \Phi_r \rangle \in P$ and $(\nu, \nu') \models \Phi_r$. (The source states of the steps were not reflected in the product states, and hence are represented using the underscore sign.) A product state was considered final iff $\mathbf{q} \in F_{\mathcal{A}^e}$ and there existed a relation $(\nu, \nu') \models \Phi_r$ such that $(\nu, \nu') \models \bigvee \{\Phi_r \mid \langle s, \Phi_r \rangle \wedge r \in F_B\}$. The antichain tree could be used for sound checking of the inclusion in this case. However, a problem was to find a subsumption relation to soundly prune the antichain tree. A natural way of defining the subsumption relation following the approach of [1] is to define the subsumption as follows: $(\langle \mathbf{q}_1, \Phi_1 \rangle, P_1) \sqsubseteq (\langle \mathbf{q}_2, \Phi_2 \rangle, P_2)$ iff (i) $\mathbf{q}_1 = \mathbf{q}_2$, (ii) $\Phi_1 \to \Phi_2$, and (iii) for each $\langle r, \Phi_r \rangle \in P_2$ there exists $\langle s, \Phi_s \rangle \in P_1$ such that $r = s$ and $\Phi_r \to \Phi_s$. Unfortunately, it turns out that using such a subsumption cannot be used for sound inclusion checking since comparing formulae representing solely the last step of the automata can lead to omitting counterexamples to inclusion that depend on longer traces. Existence of a suitable sound subsumption for this type of product states, which is needed to ensure termination of the antichain construction, is an open problem.