Verification of Data-Aware Processes via Array-Based Systems (Extended Version)

Diego Calvanese[1], Silvio Ghilardi[2], Alessandro Gianola[1], Marco Montali[1], Andrey Rivkin[1]

[1] Free University of Bozen-Bolzano
surname@inf.unibz.it

[2] Università degli Studi di Milano
silvio.ghilardi@unimi.it

ABSTRACT

We study verification over a general model of data-aware processes, to assess (parameterized) safety properties irrespectively of the initial database instance. We rely on an encoding into array-based systems, which allows us to check safety by adapting backward reachability, establishing for the first time a correspondence with model checking based on Satisfiability-Modulo-Theories (SMT). To do so, we make use of the model-theoretic machinery of model completion, which surprisingly turns out to be an effective tool for verification of relational systems, and represents the main original contribution of this paper. Our encoding pursues a twofold purpose. On the one hand, it allows us to reconstruct and generalize the essence of some of the most important decidability results obtained in the literature for artifact-centric systems, and to devise a genuinely novel class of decidable cases. On the other, it makes it possible to exploit SMT technology in implementations, building on the well-known MCMT model checker for array-based systems, and extending it to make all our foundational results fully operational.

1. INTRODUCTION

During the last two decades, a huge body of research has been dedicated to the difficult problem of reconciling master data management and business process management within contemporary organizations [40, 28, 39]. This requires to move from a purely control-flow understanding of business processes to a more holistic approach that also considers how data are manipulated and evolved by the process, and how the flow of activities is affected by the presence of data as well as the evaluation of data-driven decisions.

Two main lines of research emerged in this spectrum: one on the development of integrated models for processes and data [38], and the other on their static analysis and verification [16]. The first line of research produced a plethora of concrete languages for data-aware processes, as well as software platforms for their modeling and enactment. The main unifying theme for such approaches is a shift from standard activity-centric models to data-centric models focused on key business entities of the organization, integrating their structural and behavioral (lifecycle) aspects. This resulted in the definition of so-called object-centric [36] and artifact-centric processes [34]. In particular, the artifact-centric paradigm has been made operational by IBM, leading to the GSM (Guard-Stage-Milestone) notation [24] and the BizArtifact[1] execution platform. GSM, in turn, is the core of the CMMN OMG standard for (adaptive) case management[2].

The second line of research resulted in a series of deep, but very fragmented results on the boundaries of decidability and complexity for the static analysis of data-aware processes, considering a variety of assumptions on the model, as well as different static analysis tasks, from reachability to model checking of first-order temporal logics [44, 16]. Two main trends can be identified within this line. A recent series of results focuses on very general data-aware processes that evolve a full-fledged, relational database with arbitrary first-order constraints [11, 10, 1, 17]. Actions amount to full create-read-update-delete operations that may inject into the database fresh values taken from an infinite data domain. Verification is studied by fixing the initial instance of the database, and by considering all possible evolutions induced by the process over the initial data. This requires to verify an infinite-state transition system whose states are labeled with database instances, a problem that is highly undecidable in general. A second trend of research, with a longer tradition, is instead specifically focused on the formalization and verification of artifact-centric processes. Since the very first contributions [26, 23], the underlying formal model is based on: *(i)* a read-only relational database that stores fixed, background information, *(ii)* a working memory that stores the evolving state of artifacts, and *(iii)* actions that update the working memory.

Different variants of this model have been considered towards decidability of verification, by carefully tuning the relative expressive power of the three components. In this whole spectrum, we consider some of the most recent and intriguing approaches, ranging from pure relational structures with a single-tuple working memory [12] to artifact systems operating over a read-only database equipped with constraints, and tracking the co-evolution of multiple, unboundedly many artifacts

[1] https://sourceforge.net/projects/bizartifact/
[2] http://www.omg.org/spec/CMMN/

[27]. We do not consider numerical domains and arithmetics. Notably, the foundational results in [27] are backed up by a proof-of-concept verifier called VERIFAS [37]. Even though in these works the working memory can be updated only using values from the read-only database (i.e., no fresh values can be injected), verification is extremely challenging as it is studied parametrically to the read-only database itself, thus requiring to check infinitely many finite transition systems. This is done to assess whether the system behaves well irrespectively of the read-only data it operates on.

In this work, we propose a generalized model for artifact-centric systems that subsumes those present in the literature, in particular [12, 27, 37]. We then focus on the *(parameterized) safety problem*, which amounts to determine whether there exists an instance of the read-only database that allows the system to evolve from its initial configuration to an *undesired* configuration falsifying a given state property.

To study this problem in its full generality, we establish for the first time a bridge between verification of artifact-centric systems and model checking based on Satisfiability-Modulo-Theories (SMT). Specifically, our approach is grounded in *array-based systems*. Array-based systems are a declarative formalism originally introduced in [31, 32] to handle the verification of distributed systems, and afterwards successfully employed also to attack the static analysis of other types of systems [8, 4]. Distributed systems are parameterized in their essence: the number N of interacting processes within a distributed system is unbounded, and the challenge is that of supplying certifications that are valid for all possible values of the parameter N. The overall state of the system is typically described by means of arrays indexed by process identifiers, and used to store the content of process variables like locations and clocks. These arrays are genuine *second order* variables. In addition, *quantifiers* are used to represent sets of system states. Quantified formulae and second order function variables are at the heart of the model checking methodologies developed in [31, 32] and following papers.

The key, novel idea underlying the present paper is that of encoding artifact systems into array-based systems. This is done by providing a "functional view" of relations, where the read-only database and the artifact relations forming the working memory are represented with *sorted unary function symbols*. The resulting framework, however, requires novel and non-trivial extensions of the array-based technology to make it operational. In fact, quantifiers are handled in array-based systems both by quantifier instantiation and by quantifier elimination. Quantifier instantiation (ultimately referring to variants of the Herbrand Theorem) can be transposed to the new framework, whereas quantifier elimination becomes problematic. In fact, quantifier elimination should be applied to data variables, which do not simply range over data types (like integers, reals, or enumerated sets) as in standard array-based systems, but instead point to the content of a whole, full-fledged (read-only) relational database. To overcome this problem, we employ classic model-theoretic machinery, namely *model completions* [41]: via model completions, we prove that the runs of the systems we are interested in can be lifted (without loss of generality) to richer contexts—which we call *random-like structures*—where quantifier elimination is indeed available, despite the fact that it was not available in the original more restricted structures. This allows us to recast the original safety problem into an equivalent safety problem in the richer setting where quantifier elimination is available.

By exploiting this machinery and its model-theoretic properties, we then provide a threefold contribution:

- We consider the *backward reachability algorithm* [31, 32], one of the most widely employed techniques to effectively verify array-based systems. In particular, we show that an adaptation of this algorithm can be employed to assess safety of artifact-centric systems, retaining soundness and completeness.
- We isolate three notable classes of artifact-centric systems where the backward reachability algorithm is also guaranteed to terminate, in turn proving decidability of safety. The first two classes reconstruct and generalize the essence of the main decidability results obtained in [12] and [37], thus providing a homogeneous framework to understand them. The third class is instead novel, and to prove termination of backward reachability we resort to techniques based on well-quasi orders (in particular, a non-trivial application of Kruskal's Tree Theorem [35]).
- We build on the well-known MCMT model checker for array-based systems [33], and extend it so as to tackle verification of artifact systems. The resulting version of MCMT provides a fully operational counterpart to all the foundational results presented in the paper.

Even though implementation and experimental evaluation are not the central goal of this paper, we also note that our model checker correctly handles the examples produced to test VERIFAS [37], as well as additional examples that go beyond the verification capabilities of VERIFAS, and report some interesting case here. The performance of MCMT to conduct verification of these examples is very encouraging, and indeed provides the first stepping stone towards effective, SMT-based verification techniques for artifact-centric systems.

After giving necessary preliminaries in Section 2, we deepen our introduction to artifact-centric systems with reference to the literature in Section 3, which also provides the basis for the organization of the technical part of the paper. Full-fledged examples as well as complete proofs of all results are given in the appendix.

2. PRELIMINARIES

We adopt the usual first-order syntactic notions of signature, term, atom, (ground) formula, and so on. We use $\underline{u}$ to represent a tuple $\langle u_1, \ldots, u_n \rangle$. Our signatures Σ are multi-sorted and include equality for every sort, which implies that variables are sorted as well. Depending on the context, we keep the sort of a variable implicit, or we indicate explicitly in a formula that variable x has sort S by employing notation $x : S$. The

notation $t(\underline{x})$, $\phi(\underline{x})$ means that the term t, the formula ϕ has free variables included in the tuple $\underline{x}$. We are concerned only with constants and function symbols f, each of which has *sources* $\underline{S}$ and a *target* S', denoted as $f : \underline{S} \longrightarrow S'$. We assume that terms and formulae are well-typed, in the sense that the sorts of variables, constants, and function sources/targets match. A formula is said to be *universal* (resp., *existential*) if it has the form $\forall \underline{x}\,(\phi(\underline{x}))$ (resp., $\exists \underline{x}\,(\phi(\underline{x}))$), where ϕ is a quantifier-free formula. Formulae with no free variables are called *sentences*.

From the semantic side, we use the standard notions of a Σ-*structure* $\mathcal{M}$ and of *truth* of a formula in a Σ-structure under an assignment to the free variables. A Σ-*theory* T is a set of Σ-sentences; a *model* of T is a Σ-structure $\mathcal{M}$ where all sentences in T are true. We use the standard notation $T \models \phi$ to say that ϕ is true in all models of T for every assignment to the free variables of ϕ. We say that ϕ is T-*satisfiable* iff there is a model $\mathcal{M}$ of T and an assignment to the free variables of ϕ that make ϕ true in $\mathcal{M}$.

In the following, we use definable extensions as a means to introduce case-defined functions F, abbreviating more complicated (still first-order) expressions. Let us fix a signature Σ and a Σ-theory T; a T-*partition* is a finite set $\kappa_1(\underline{x}), \ldots, \kappa_n(\underline{x})$ of quantifier-free formulae such that $T \models \forall \underline{x} \bigvee_{i=1}^{n} \kappa_i(\underline{x})$ and $T \models \bigwedge_{i \neq j} \forall \underline{x} \neg(\kappa_i(\underline{x}) \wedge \kappa_j(\underline{x}))$. Given such a T-partition $\kappa_1(\underline{x}), \ldots, \kappa_n(\underline{x})$ together with Σ-terms $t_1(\underline{x}), \ldots, t_n(\underline{x})$ (all of the same target sort), a *case-definable extension* is the Σ'-theory T', where $\Sigma' = \Sigma \cup \{F\}$, with F a "fresh" function symbol (i.e., $F \notin \Sigma$)[3], and $T' = T \cup \bigcup_{i=1}^{n} \{\forall \underline{x}\,(\kappa_i(\underline{x}) \rightarrow F(\underline{x}) = t_i(\underline{x}))\}$. Intuitively, F represents a case-defined function, which can be reformulated using nested if-then-else expressions and can be written as

$$F(\underline{x}) \;:=\; \texttt{case of}\ \{\kappa_1(\underline{x}) : t_1; \cdots ; \kappa_n(\underline{x}) : t_n\}.$$

By abuse of notation, we shall identify T with any of its case-definable extensions T'. In fact, it is easy to produce from a Σ'-formula ϕ' a Σ-formula ϕ that is equivalent to ϕ' in all models of T': just remove (in the appropriate order) every occurrence $F(\underline{v})$ of the new symbol F in an atomic formula A, by replacing A with $\bigvee_{i=1}^{n}(\kappa_i(\underline{v}) \wedge A(t_i(\underline{v})))$. We also exploit λ-abstractions (see, e.g., formula (6) below) for more "compact" representation of some complex expressions, and always use them in atoms like $b = \lambda y.F(y, \underline{z})$ as abbreviations of $\forall y.\ b(y) = F(y, \underline{z})$ (where, typically, F is a symbol introduced in a case-defined extension as above). Thus, also formula containing lambda abstractions, can be converted into plain first-order formulae.

3. ARTIFACT SYSTEMS

To capture data-aware processes, we follow the traditional line of research focused on the formal representation of *artifact systems*. Since their initial versions [26, 23], such systems are traditionally formalized using three components: *(i) a read-only database (DB)*, storing background information that does *not* change during the system evolution; *(ii) an artifact working memory*, storing data and lifecycle information about artifact(s) that *does* change during the system evolution; *(iii) actions* (also called *services*) that access the read-only database and the working memory, and determine how the working memory itself has to be updated.

Different variants of this framework have been considered towards decidability of verification, by carefully tuning the expressive power of the three components. As for the read-only DB, approaches differ depending on which constraints may be attached to the DB relation schemas. Results range from the basic case where the DB schema consists of a purely relational structure without constraints [12], to the case where (restricted forms) of keys and foreign keys are supported [27].

As for the working memory, radically different models are obtained depending on whether only a single artifact instance is evolved, or whether instead the coevolution of multiple instances of possibly different artifacts is supported. In particular, early formal models for artifact systems merely considered a fixed set of so-called *artifact variables*, altogether instantiated into a single tuple of data. This, in turn, allows to capture the evolution of a single artifact instance. This is the model studied in [26, 12]. We call artifact systems of this form *Simple Artifact System (SAS)*. More sophisticated types of artifact systems have been instead recently studied in [27, 37]. Here, the working memory is not only equipped with artifact variables as in SAS, but also with so-called *artifact relations*, which supports storing arbitrarily many tuples, each accounting for a different artifact instance that can be separately evolved on its own. We call artifact systems of this form *Relational Artifact System (RAS)*.

Finally, actions are usually specified using quantifier-free formulae relating the content of the read-only DB as well as the current configuration of the working memory to (possibly different) next configurations. An applicable action may be executed, nondeterministically transforming the current configuration of working memory in one of such next configurations. When multiple artifact instances can be evolved, the shape of actions has to be controlled towards decidability of verification, especially by limiting the ability of a single action to update *multiple relations at once*, as well as that of *comparing the content of different relations* [27, 37]. In addition, actions typically *do not bring in new data* while updating the working memory, but only use values present in the read-only DB. The only exception to this is [23], which requires to insert possibly fresh numerical values due to arithmetics, which we do not consider here.

Starting from this basis, we study a more general model of artifact systems accounting for various forms of updates (possibly comparing and updating multiple relations at once), and for the injection of possibly fresh values during the execution. Specifically, we study read-only DB schemas in Section 4, SAS in Section 5, and RAS in Section 6. We consider the *safety problem* for such systems, which amounts to check whether there

[3]Arity, source sorts and target sort for F can be deduced from the context (considering that everything is well-typed).

exists an instance of the read-only DB such that the system can evolve from the initial configuration of its working memory to an *undesired* configuration. Since in the case of RAS the decidability of verification subtly depends the richness of actions in updating the working memory, in Section 7 we study kinds of actions and of the read-only DB schema that guarantee termination.

4. READ-ONLY DATABASE SCHEMAS

We now provide a formal definition of (read-only) DB-schemas by relying on an algebraic, functional characterization, and derive some key model-theoretic properties instrumental to the technical treatment.

Definition 1. A *DB schema* is a pair $\langle \Sigma, T \rangle$, where: *(i)* Σ is a *DB signature*, that is, a finite multi-sorted signature whose only symbols are equality, unary functions, and constants; *(ii)* T is a *DB theory*, that is, a set of universal Σ-sentences. $\triangleleft$

Next, we refer to a DB schema simply through its (DB) theory T and (DB) signature Σ. Given a DB signature Σ, we denote by Σ_{srt} the set of sorts and by Σ_{fun} the set of functions in Σ. Since Σ contains only unary function symbols and equality, all atomic Σ-formulae are of the form $t_1(v_1) = t_2(v_2)$, where t_1, t_2 are possibly complex terms, and v_1, v_2 are either variables or constants.

We associate to a DB signature Σ a characteristic graph $G(\Sigma)$ capturing the dependencies induced by functions over sorts. Specifically, $G(\Sigma)$ is an edge-labeled graph whose set of nodes is Σ_{srt}, and with a labeled edge $S \xrightarrow{f} S'$ for each $f : S \longrightarrow S'$ in Σ_{fun}. We say that Σ is *acyclic* if $G(\Sigma)$ is so. The *leaves* of Σ are the nodes of $G(\Sigma)$ without outgoing edges. These terminal sorts are divided in two subsets, respectively representing *unary relations* and *value sorts*. Non-value sorts (i.e., unary relations and non-leaf sorts) are called *id sorts*, and are conceptually used to represent (identifiers of) different kinds of objects. Value sorts, instead, represent datatypes such as strings, numbers, clock values, etc. We denote the set of id sorts in Σ by Σ_{ids}, and that of value sorts by Σ_{val}, hence $\Sigma_{srt} = \Sigma_{ids} \uplus \Sigma_{val}$.

We now consider extensional data.

Definition 2. A *DB instance* of DB schema $\langle \Sigma, T \rangle$ is a Σ-structure $\mathcal{M}$ that is a model of T and such that every id sort of Σ is interpreted in $\mathcal{M}$ on a *finite* set. $\triangleleft$

Contrast this to arbitrary *models* of T, where no finiteness assumption is made. What may appear as not customary in Definition 2 is the fact that value sorts can be interpreted on infinite sets. This allows us, at once, to reconstruct the classical notion of DB instance as a finite model (since only finitely many values can be pointed from id sorts using functions), at the same time supplying a potentially infinite set of fresh values to be dynamically introduced in the working memory during the evolution of the artifact system. More details on this will be given in Section 4.1.

We respectively denote by $S^{\mathcal{M}}$, $f^{\mathcal{M}}$, and $c^{\mathcal{M}}$ the interpretation in $\mathcal{M}$ of the sort S (this is a set), of the function symbol f (this is a set-theoretic function), and of the constant c (this is an element of the interpretation of the corresponding sort). Obviously, $f^{\mathcal{M}}$ and $c^{\mathcal{M}}$ must match the sorts in Σ. E.g., if f has source S and target U, then $f^{\mathcal{M}}$ has domain $S^{\mathcal{M}}$ and range $U^{\mathcal{M}}$.

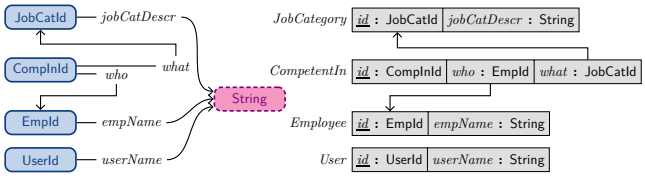


Figure 1: On the left: characteristic graph of the human resources DB signature from Example 1. On the right: relational view of the DB signature; each cell denotes an attribute with its type, underlined attributes denote primary keys, and directed edges capture foreign keys.

Example 1. The human resource (HR) branch of a company stores the following information inside a relational database: *(i)* users registered to the company website, who are potentially interested in job positions offered by the company; *(ii)* the different, available job categories; *(iii)* employees belonging to HR, together with the job categories they are competent in (in turn indicating which job applicants they could interview). To formalize these different aspects, we make use of a DB signature Σ_{hr} consisting of: *(i)* four id sorts, used to respectively identify users, employees, job categories, and the competence relationship connecting employees to job categories; *(ii)* one value sort containing strings used to name users and employees, and describe job categories. In addition, Σ_{hr} contains five function symbols mapping: *(i)* user identifiers to their corresponding names; *(ii)* employee identifiers to their corresponding names; *(iii)* job category identifiers to their corresponding descriptions; *(iv)* competence identifiers to their corresponding employees and job categories. The characteristic graph of Σ_{hr} is shown in Figure 1 (left part). $\triangleleft$

We close the formalization of DB schemas by discussing DB theories. The role of a DB theory is to encode background axioms to express constraints on the different elements of the corresponding signature. We illustrate a typical background axiom, required to handle the possible presence of *undefined identifiers/-values* in the different sorts. This, in turn, is essential to capture artifact systems whose working memory is initially undefined, in the style of [27, 37]. To accommodate this, to specify an undefined value we add to every sort S of Σ a constant $\texttt{undef}_S$ (written from now on, by abuse of notation, just as $\texttt{undef}$, used also to indicate a tuple). Then, for each function symbol f of Σ, we add the following axiom to the DB theory:

$$\forall x \; (x = \texttt{undef} \leftrightarrow f(x) = \texttt{undef}) \qquad (1)$$

This axiom states that the application of f to the undefined value produces an undefined value, and it is the only situation for which f is undefined.

Remark 1. *In the following, we always implicitly as-*

sume that the DB theory consists of Axioms 1, but our technical results are not bound to this specific choice. The specific conditions we require on the DB theory towards our results will be explicitly stated later. ◁

4.1 Relational View of DB Schemas

We now clarify how the algebraic, functional characterization of DB schema and instance can be actually reinterpreted in the classical, relational model. Definition 1 naturally corresponds to the definition of relational database schema equipped with single-attribute *primary keys* and *foreign keys* (plus a reformulation of constraint (1)). To technically explain the correspondence, we adopt the *named perspective*, where each relation schema is defined by a signature containing a *relation name* and a set of *typed attribute names*. Let $\langle \Sigma, T \rangle$ be a DB schema. Each id sort $S \in \Sigma_{ids}$ corresponds to a dedicated relation R_S with the following attributes: *(i)* one identifier attribute id_S with type S; *(ii)* one dedicated attribute a_f with type S' for every function symbol $f \in \Sigma_{fun}$ of the form $f : S \longrightarrow S'$.

The fact that R_S is built starting from functions in Σ naturally induces different database dependencies in R_S. In particular, for each non-id attribute a_f of R_S, we get a *functional dependency* from id_S to a_f; altogether, such dependencies in turn witness that id_S is the *(primary) key* of R_S. In addition, for each non-id attribute a_f of R_S whose corresponding function symbol f has id sort S' as image, we get an *inclusion dependency* from a_f to the id attribute $id_{S'}$ of $R_{S'}$; this captures that a_f is a *foreign key* referencing $R_{S'}$.

Example 2. The diagram on the right in Figure 1 graphically depicts the relational view corresponding to the DB signature of Example 1. ◁

Given a DB instance $\mathcal{M}$ of $\langle \Sigma, T \rangle$, its corresponding relational instance $\mathcal{I}$ is the minimal set satisfying the following property: for every id sort $S \in \Sigma_{ids}$, let $f_1, \ldots, f_n$ be all functions in Σ with domain S; then, for every identifier $\mathtt{o} \in S^{\mathcal{M}}$, $\mathcal{I}$ contains a *labeled fact* of the form $R_S(id_S : \mathtt{o}^{\mathcal{M}}, a_{f_1} : f_1^{\mathcal{M}}(\mathtt{o}^{\mathcal{M}}), \ldots, a_{f_n} : f_n^{\mathcal{M}}(\mathtt{o}^{\mathcal{M}}))$. With this interpretation, the active domain of $\mathcal{I}$ is the set

$$\bigcup_{S \in \Sigma_{ids}} (S^{\mathcal{M}} \setminus \{\mathtt{undef}^{\mathcal{M}}\}) \cup \left\{ \mathtt{v} \in \bigcup_{V \in \Sigma_{val}} V^{\mathcal{M}} \;\middle|\; \begin{array}{l} \text{there exist } f \in \Sigma_{fun} \\ \text{and } \mathtt{o} \in dom(f^{\mathcal{M}}) \text{ s.t. } f^{\mathcal{M}}(\mathtt{o}) = \mathtt{v} \end{array} \right\}$$

consisting of all (proper) identifiers assigned by $\mathcal{M}$ to id sorts, as well as values obtained in $\mathcal{M}$ via the application of some function. Since such values are necessarily *finitely many*, one may wonder why in Definition 2 we allow for interpreting value sorts over infinite sets. The reason is that, in our framework, an evolving artifact system may use such infinite provision to inject and manipulate new values into the working memory.

This relational interpretation of DB schemas exactly reconstruct the requirements posed by [27, 37] on the schema of the *read-only* database: *(i)* each relation schema has a single-attribute primary key; *(ii)* attributes are typed; *(iii)* attributes may be foreign keys referencing other relation schemas; *(iv)* the primary keys of different relation schemas are pairwise disjoint.

We stress that all such requirements are natively captured in our functional definition of a DB signature, and do not need to be formulated as axioms in the DB theory. The DB theory is used to express additional constraints, like that in Axiom (1). In the following section, we thoroughly discuss which properties must be respected by signatures and theories to guarantee that our verification machinery is well-behaved.

One may wonder why we have not directly adopted a relational view for DB schemas. This will become clear during the technical development. We anticipate the main, intuitive reasons. First, our functional view allows us to reconstruct in a single, homogeneous framework, some important results on verification of artifact systems, achieved on different models that have been unrelated so far [12, 27]. E.g., the model adopted in [12] cannot be naively extended with keys, since relational structures with key constraints do not enjoy the amalgamation property, which is the crucial condition for the main decidability results in that work. Second, our functional view makes the dependencies among different types explicit. In fact, our notion of characteristic graph, which is readily computed from a DB signature, exactly reconstructs the central notion of foreign key graph used in [27] towards the main decidability results.

4.2 Formal Properties of DB Schemas

The theory T from Definition 1 must satisfy few crucial requirements for our approach to work. In this section, we define such requirements and show that they are matched whenever the signature Σ is acyclic. Actually, acyclicity is a stronger requirement than needed, which, however, simplifies our exposition.

4.2.1 Finite model property

A Σ-formula ϕ is a Σ-*constraint* (or just a constraint) iff it is a conjunction of literals. The constraint satisfiability problem for T asks: given an existential formula $\exists \underline{y}\, \phi(\underline{x}, \underline{y})$ (with ϕ a constraint[4]), are there a model $\mathcal{M}$ of T and an assignment α to the free variables $\underline{x}$ s.t. $\mathcal{M}, \alpha \models \exists \underline{y}\, \phi(\underline{x}, \underline{y})$?

We say that T has the *finite model property* (for constraint satisfiability) iff every constraint ϕ that is satisfiable in a model of T is satisfiable in a DB instance of T.[5] The following is proved in Appendix B:

Proposition 1. *T has the finite model property in case Σ is acyclic.* ◁

The finite model property implies decidability of the constraint satisfiability problem in case T is recursively axiomatized.

4.2.2 Quantifier elimination

A Σ-theory T has *quantifier elimination* iff for every

[4] For the purposes of this definition, we may equivalently take ϕ to be quantifier-free.

[5] It is easily seen that this implies that ϕ is satisfiable also in a DB instance interpreting also value sorts into finite sets.

Σ-formula $\phi(\underline{x})$ there is a quantifier-free formula $\phi'(\underline{x})$ such that $T \models \phi(\underline{x}) \leftrightarrow \phi'(\underline{x})$. It is known that quantifier elimination holds if quantifiers can be eliminated from *primitive* formulae, i.e., formulae of the kind $\exists \underline{y}\, \phi(\underline{x}, \underline{y})$, with ϕ a constraint. We assume that when quantifier elimination is considered, there is an effective procedure that eliminates quantifiers.

A DB theory T does not necessarily have quantifier elimination; it is however often possible to strengthen T in a conservative way (w.r.t. constraint satisfiability) and get quantifier elimination. We say that T has a *model completion* iff there is a stronger theory $T^* \supseteq T$ (still within the same signature Σ of T) s.t. (i) every Σ-constraint satisfiable in a model of T is also so in a model of T^*; (ii) T^* has quantifier elimination. T^* is called a *model completion* of T.

PROPOSITION 2. *T has a model completion in case it is axiomatized by universal one-variable formulae and Σ is acyclic.* ◁

In Appendix B we prove the above proposition and give an algorithm for quantifier elimination. This algorithm is far from optimal from two points of view. First, contrary to what happens in linear arithmetics, the quantifier elimination needed to prove Proposition 2 has a much better behaviour (from the complexity point of view) if obtained via a suitable version of the Knuth-Bendix procedure [9]. Since these aspects concerning quantifier elimination are rather delicate, we address them in a dedicated paper [18] (our MCMT implementation, however, already partially takes into account such future development).

Secondly, the algorithm presented in Appendix B uses the acyclicity assumption, whereas such assumption is in general not needed for Proposition 2 to hold: for instance, when $T := \emptyset$ or when T contains only Axiom (1), the model completion can be proved to exist, even if Σ is not acyclic, by using the Knuth-Bendix version of the quantifier elimination algorithm.

Hereafter, we make the following assumption:

ASSUMPTION 1. *The DB theories we consider have decidable constraint satisfiability problem, finite model property, and admit a model completion.* ◁

This assumption is matched, for instance, in the following three cases: *(i)* when Σ is acyclic; *(ii)* when T is empty; *(iii)* when T is axiomatized by Axiom (1).

5. SIMPLE ARTIFACT SYSTEMS

In this section we consider systems manipulating only individual variables and reading data from a given database instance. In order to introduce verification problems in a symbolic setting, one first has to specify which formulae are used to represent sets of states, the system initializations, and system evolution. Given a DB schema $\langle \Sigma, T \rangle$ and a tuple $\underline{x} = x_1, \ldots, x_n$ of variables, we introduce the following classes of Σ-formulae:

- a *state formula* is a quantifier-free Σ-formula $\phi(\underline{x})$;
- an *initial formula* is a conjunction of equalities of the form $\bigwedge_{i=1}^{n} x_i = c_i$, where each c_i is a constant;[6]
- a *transition formula* $\hat{\tau}$ is an existential formula

$$\exists \underline{y}\, \big(G(\underline{x}, \underline{y}) \wedge \textstyle\bigwedge_{i=1}^{n} x_i' = F_i(\underline{x}, \underline{y})\big) \qquad (2)$$

where $\underline{x}'$ are renamed copies of $\underline{x}$, G is quantifier-free and $F_1, \ldots, F_n$ are case-defined functions. We call G the *guard* and F_i the *updates* of Formula (2).

DEFINITION 3. A *Simple Artifact System* (SAS) is

$$\mathcal{S} \;=\; \langle \Sigma, T, \underline{x}, \iota(\underline{x}), \tau(\underline{x}, \underline{x}') \rangle$$

where: *(i)* $\langle \Sigma, T \rangle$ is a (read-only) DB schema, *(ii)* $\underline{x} = x_1, \ldots, x_n$ are variables (called *artifact variables*), *(iii)* ι is an initial formula, and *(iv)* τ is a disjunction of transition formulae. ◁

EXAMPLE 3. We consider a SAS working over the DB schema of Example 1. It captures a global, single-instance artifact tracking the main, overall phases of a hiring process. The job hiring artifact employs a dedicated *pState* variable to store the current process state. Initially, hiring is disabled, which is captured by setting the *pState* variable to `undef`. A transition of the process from disabled to *enabled* may occur provided that the read-only HR DB contains at least one registered user (who, in turn, may decide to apply for a job). Technically, we introduce a dedicated artifact variable *uId* initialized to `undef`, and used to load the identifier of such a registered user, if (s)he exists. Enablement is then captured by the following transition formula:

$$\exists y : \mathsf{UserId} \begin{pmatrix} pState = \texttt{undef} \wedge y \neq \texttt{undef} \\ \wedge\, pState' = \texttt{enabled} \wedge uId' = y \end{pmatrix}$$

Notice in particular how the existence of a user is checked using the typed variable y, checking that it is not `undef` and correspondingly assigning it to *uId*. ◁

A *safety* formula for SAS $\mathcal{S}$ is a state formula $\upsilon(\underline{x})$ describing undesired states of $\mathcal{S}$. We say that $\mathcal{S}$ is *safe* with respect to υ if intuitively the system has no finite run leading from ι to υ. Formally, there is no DB-instance $\mathcal{M}$ of $\langle \Sigma, T \rangle$, no $k \geq 0$, and no assignment in $\mathcal{M}$ to the variables $\underline{x}^0, \ldots, \underline{x}^k$ such that the formula

$$\iota(\underline{x}^0) \wedge \tau(\underline{x}^0, \underline{x}^1) \wedge \cdots \wedge \tau(\underline{x}^{k-1}, \underline{x}^k) \wedge \upsilon(\underline{x}^k) \qquad (3)$$

is true in $\mathcal{M}$ (here $\underline{x}^i$'s are renamed copies of $\underline{x}$). The safety problem for $\mathcal{S}$ is the following: *given a safety formula υ decide whether $\mathcal{S}$ is safe w.r.t. υ.*

Algorithm 1 describes the *backward reachability algorithm* (or, *backward search*) for handling the safety problem for $\mathcal{S}$. An integral part of the algorithm is to compute preimages. For that purpose, we define for any $\phi_1(\underline{z}, \underline{z}')$ and $\phi_2(\underline{z})$, $Pre(\phi_1, \phi_2)$ as the formula $\exists \underline{z}'(\phi_1(\underline{z}, \underline{z}') \wedge \phi_2(\underline{z}'))$. The *preimage* of the set of states described by a state formula $\phi(\underline{x})$ is the set of states described by $Pre(\tau, \phi)$.[7] The subprocedure $\mathsf{QE}(T^*, \phi)$ in Line 6 applies the quantifier elimination algorithm of T^* to the existential formula ϕ. Algorithm 1 computes iterated preimages of υ and applies to them quantifier elimination, until a fixpoint is reached or until a set

[6] Typically, c_i is an `undef` constant mentioned above.
[7] Notice that, when $\tau = \bigvee \hat{\tau}$, then $Pre(\tau, \phi) = \bigvee Pre(\hat{\tau}, \phi)$.

Algorithm 1: Backward reachability algorithm

```
   Function BReach(υ)
1  |  φ ⟵ υ; B ⟵ ⊥;
2  |  while φ ∧ ¬B is T-satisfiable do
3  |  |  if ι ∧ φ is T-satisfiable. then
   |  |  |  return unsafe
4  |  |  B ⟵ φ ∨ B;
5  |  |  φ ⟵ Pre(τ, φ);
6  |  |  φ ⟵ QE(T*, φ);
   |  |  return (safe, B);
```

intersecting the initial states (i.e., satisfying ι) is found.

We state now the main result of this section:

THEOREM 1. *Let $\langle \Sigma, T\rangle$ be a DB schema. Then, for every SAS $\mathcal{S} = \langle \Sigma, T, \underline{x}, \iota, \tau\rangle$ the following holds: (1) backward search is effective and partially correct for solving safety problems for $\mathcal{S}$;*[8] *(2) if Σ is acyclic, backward search terminates and decides safety problems for $\mathcal{S}$ in* PSPACE *in the combined size of $\underline{x}$, ι, and τ.* ◁

PROOF (SKETCH). Algorithm 1, to be effective, requires the availability of decision procedures for discharging the satisfiability tests in Lines 2-3. Thanks to our hypotheses in Assumption 1, we can freely assume that all the runs we are interested in take place inside models of T^* where we can eliminate quantifiers: in fact, formulae of the kind (3) are existential so they are satisfiable in a model of T iff they are satisfiable in a DB instance iff they are satisfiable in a model of T^*. Thanks to quantifier elimination, the preimage of a state is a state formula and this fact is exploited both to make safety and fixpoint tests effective, and to ensure termination (because there are only finitely many state formulae, up to T-equivalence). As for complexity, it can be shown that, when Σ is acyclic, backward search can be modified so as to run in PSPACE. □

The proof of Theorem 1 shows that, whenever Σ is not acyclic, backward search is still a semi-decision procedure: if the system is unsafe, backward search always terminates and discovers it; if the system is safe, the procedure can diverge (but it is still correct).

Theorem 1 reconstructs the main decidability and complexity result from [12], restricted to first-order definable classes of database instances used in our case. First, it can be shown that every existential formula $\phi(\underline{x}, \underline{x}')$ can be turned into the form of Formula (2). The proof of Statement (2) of Theorem 1 requires that T: *(i)* admits a model completion; *(ii)* is *locally finite*, i.e., up to T-equivalence, there are only finitely many atoms involving a fixed finite number of variables (this condition is assumed in [12], and is implied by acyclicity); *(iii)* is universal; and *(iv)* enjoys decidability of constraint satisfiability. Conditions *(iii)* and *(iv)* imply that one can decide whether a finite structure is a model of T (as assumed in [12]). If *(ii)* and *(iii)* hold, it is well-known that *(i)* is equivalent to amalgamation [45]. Moreover, *(ii)* alone always holds for relational signatures and *(iii)* is equivalent to T being closed under substructures (this is a standard preservation theorem in model theory [21]). It follows that relational signatures (and locally finite theories in general) require only amalgamability and closure under substructures, which precisely coincide with the hypotheses in [12].

The major difference with [12] is in the adopted system settings. In our first-order case we can perform verification in a *purely symbolic* way, using (semi-)decision procedures provided by SMT-solvers, even when local finiteness fails. As mentioned before, local finiteness is guaranteed in the relational context, but it does not hold anymore when *arithmetic operations* are introduced. Note that the theory of a single uninterpreted binary relation (i.e., the theory of graphs) is amalgamable, whereas it can be easily seen that the theory of one binary relation endowed with primary key dependencies is not. Our second distinctive feature naturally follows from this observation: thanks to our functional representation of DB schemas (with keys), the amalgamation property, required by Theorem 3, holds.

6. RELATIONAL ARTIFACT SYSTEMS

We now turn to systems manipulating higher order variables, which are supposed to model evolving relations, the so-called "artifact relations". The idea is to treat artifact relations in a uniform way as we did for the the read-only DB: we need extra sort symbols (recall that each sort symbol corresponds to a database relation symbol) and extra unary function symbols, the latter being treated as second-order variables.

Given a DB schema Σ, an *artifact extension* of Σ is a signature Σ_{ext} obtained from Σ by adding to it some extra sort symbols[9]. These new sorts (usually indicated with letters $E, F, \dots$) are called *artifact sorts* (or *artifact relations* by some abuse of terminology), while the old sorts from Σ are called *basic sorts*. Below, given $\langle \Sigma, T\rangle$ and an artifact extension Σ_{ext} of Σ, when we speak of a Σ_{ext}-model of T, a DB instance of $\langle \Sigma_{ext}, T\rangle$, or a Σ_{ext}-model of T^*, we mean a Σ_{ext}-structure $\mathcal{M}$ whose reduct to Σ respectively is a model of T, a DB instance of $\langle \Sigma, T\rangle$, or a model of T^*.

An *artifact setting* over Σ_{ext} is a pair $(\underline{x}, \underline{a})$ given by a finite set of individual variables $\underline{x}$ and a finite set of unary function variables $\underline{a}$: *the latter are required to have an artifact sort as source sort and a basic sort as target sort.* Variables in $\underline{x}$ are called (as before) *artifact variables*, and variables in $\underline{a}$ *artifact components*.

Given a DB instance $\mathcal{M}$ of Σ_{ext}, an *assignment* to an artifact setting $(\underline{x}, \underline{a})$ over Σ_{ext} is a map α assigning to every artifact variable $x_i \in \underline{x}$ of sort S_i an element $x^\alpha \in S_i^{\mathcal{M}}$ and to every artifact component $a_j : E_j \longrightarrow U_j$ (with $a_j \in \underline{a}$) a set-theoretic function $a_j^\alpha : E_j^{\mathcal{M}} \longrightarrow U_j^{\mathcal{M}}$.

[8] Partial correctness means that, when the algorithm terminates, it gives a correct answer. Effectiveness means that all subprocedures in the algorithm can be effectively executed.

[9] By 'signature' we always mean 'signature with equality', so as soon as new sorts are added, the corresponding equality predicates are added too.

We can view an assignment to an artifact setting $(\underline{x}, \underline{a})$ as a DB instance *extending* the DB instance $\mathcal{M}$ as follows. Let all the artifact components in $(\underline{x}, \underline{a})$ having source E be $a_{i_1} : E \longrightarrow S_1, \cdots, a_{i_n} : E \longrightarrow S_n$. Viewed as a relation in the artifact assignment $(\mathcal{M}, \alpha)$, the artifact relation E "consists" of the set of tuples

$$\{\langle e, a^{\alpha}_{i_1}(e), \ldots, a^{\alpha}_{i_n}(e)\rangle \mid e \in E^{\mathcal{M}}\}$$

Thus each element of E is formed by an "entry" $e \in E^{\mathcal{M}}$ (uniquely identifying the tuple) and by "data" $\underline{a}^{\alpha}_i(e)$ taken from the read-only database $\mathcal{M}$. When the system evolves, the set $E^{\mathcal{M}}$ of entries remains fixed, whereas the components $\underline{a}^{\alpha}_i(e)$ may change: typically, we initially have $\underline{a}^{\alpha}_i(e) = \texttt{undef}$, but these values are changed when some defined values are inserted into the relation modeled by E; the values are then repeatedly modified (and possibly also reset to `undef`, if the tuple is removed and e is re-set to point to undefined values)[10].

To introduce Relational Artifact Systems we discuss the kind of formulae we use. In such formulae, we use notations like $\phi(\underline{z}, \underline{a})$ to mean that ϕ is a formula whose free individual variables are among the $\underline{z}$ and whose free unary function variables are among the $\underline{a}$. Let $(\underline{x}, \underline{a})$ be an artifact setting over Σ_{ext}, where $\underline{x} = x_1, \ldots, x_n$ are the artifact variables and $\underline{a} = a_1, \ldots, a_m$ are the artifact components (their source and target sorts are left implicitly specified):

- An *initial formula* is a formula $\iota(\underline{x})$ of the form[11]
$$\textstyle\bigwedge_{i=1}^{n} x_i = c_i \wedge \bigwedge_{j=1}^{m} a_j = \lambda y.d_j \tag{4}$$
where c_i, d_j are constants from Σ.
- A *state formula* has the form
$$\exists \underline{e}\, \phi(\underline{e}, \underline{x}, \underline{a}) \tag{5}$$
where ϕ is quantifier-free and the $\underline{e}$ are individual variables of artifact sorts.
- A *transition formula* $\hat{\tau}$ has the form
$$\exists \underline{e} \begin{pmatrix} \gamma(\underline{e}, \underline{x}, \underline{a}) \wedge \bigwedge_i x'_i = F_i(\underline{e}, \underline{x}, \underline{a}) \\ \wedge \bigwedge_j a'_j = \lambda y.G_j(y, \underline{e}, \underline{x}, \underline{a}) \end{pmatrix} \tag{6}$$
where the $\underline{e}$ are individual variables (of *both* basic and artifact sorts), γ (the 'guard') is quantifier-free, $\underline{x}'$, $\underline{a}'$ are renamed copies of $\underline{x}$, $\underline{a}$, and the F_i, G_j (the 'updates') are case-defined functions.

Note that transition formulae as above can express, e.g., *(i)* insertion (with/without duplicates) of a tuple in an artifact relation, *(ii)* removal of a tuple from an artifact relation, *(iii)* transfer of a tuple from an artifact relation to artifact variables (and vice versa), and *(iv)* removal/modification of *all* the tuples satisfying a certain condition from an artifact relation. All the above operations can also be constrained. Our framework is strictly more expressive than, e.g., the one in [37], as shown in Appendix F, and also than the one in [27].

Definition 4. A *Relational Artifact System* (RAS) is

$$\mathcal{S} = \langle \Sigma, T, \Sigma_{ext}, \underline{x}, \underline{a}, \iota(\underline{x}, \underline{a}), \tau(\underline{x}, \underline{a}, \underline{x}', \underline{a}')\rangle$$

where: *(i)* $\langle \Sigma, T\rangle$ is a (read-only) DB schema, *(ii)* Σ_{ext} is an artifact extension of Σ, *(iii)* $(\underline{x}, \underline{a})$ is an artifact setting over Σ_{ext}, *(iv)* ι is an intitial formula, and *(v)* τ is a disjunction of transition formulae. ◁

Example 4. We transform the SAS of Example 3 into a RAS $\mathcal{S}_{hr}$ containing a multi-instance artifact accounting for the evolution of *job applications*. Each job category may receive multiple applications from registered users. Such applications are then evaluated, finally deciding which are accepted and which are rejected. The example is inspired by the job hiring process presented in [43] to show the intrinsic difficulties of capturing real-life processes with many-to-many interacting business entities using conventional process modeling notations (such as BPMN). An extended version of this example, capturing the co-evolution of multiple instances of two different artifacts, is presented in Appendix A.1.

As for the read-only DB, $\mathcal{S}_{hr}$ works over the DB schema of Example 1, extended with a further value sort Score used to score job applications. Score contains 102 values in the range $[-1, 100]$, where -1 denotes the non-eligibility of the application, and a score from 0 to 100 indicates the actual one assigned after evaluating the application. For the sake of readability, we use usual predicates $<$, $>$ and $=$ to compare variables of type Score. This is syntactic sugar and does not require to introduce rigid predicates in our framework.

As for the working memory, $\mathcal{S}_{hr}$ consists of two artifacts: the single-instance *job hiring* artifact tracking the three main phases of the overall process (and described in Example 3), and a multi-instance artifact accounting for the evolution of *user applications*. To model applications, we take the DB signature Σ_{hr} of the read-only database of human resources, and enrich it with an artifact extension containing an artifact sort AppIndex used to *index* (i.e., *"internally" identify*) job applications. The management of job applications is then modeled by an artifact setting with: *(i)* artifact components with domain AppIndex capturing the artifact relation that store the different job applications; *(ii)* additional individual variables as a temporary memory to manipulate the artifact relation. Specifically, each application consists of a job category, the identifier of the applicant user and that of an HR employee responsible for the application, the application score and final result (indicating whether the application is among the winners or the losers for the job offer). These information slots are encapsulated into dedicated artifact components, i.e., function variables with domain AppIndex that collectively realize the application artifact relation:

$$\begin{array}{lcl} \mathit{appJobCat} & : & \mathsf{AppIndex} \longrightarrow \mathsf{JobCatId} \\ \mathit{applicant} & : & \mathsf{AppIndex} \longrightarrow \mathsf{UserId} \\ \mathit{appResp} & : & \mathsf{AppIndex} \longrightarrow \mathsf{EmpId} \\ \mathit{appScore} & : & \mathsf{AppIndex} \longrightarrow \mathsf{Score} \\ \mathit{appResult} & : & \mathsf{AppIndex} \longrightarrow \mathsf{String} \end{array}$$

We now discuss the relevant transitions for inserting and evaluating job applications. When writing transition formulae, we make the following assumption: if an artifact variable/component is not mentioned at all, it

[10] In accordance with MCMT conventions, we denote the application of an artifact component a to a term (i.e., constant or variable) v also as $a[v]$, instead of $a(v)$.

[11] Recall that $a_j = \lambda y.d_j$ abbreviates $\forall y\, a_j(y) = d_j$.

is meant that is updated identically; otherwise, the relevant update function will specify how it is updated.[12] The insertion of an application into the system can be executed when the hiring process is enabled (cf. Example 3), and consists of two consecutive steps. To indicate when a step can be applied, also ensuring that the insertion of an application is not interrupted by the insertion of another one, we manipulate a string artifact variable *aState*. The first step is executable when *aState* is `undef`, and aims at loading the application data into dedicated artifact variables through the following simultaneous effects: *(i)* the identifiers of the user who wants to submit the application, and that of the targeted job category, are selected and respectively stored into variables *uId* and *jId*; *(ii)* the identifier of an HR employee who becomes responsible for the application is selected and stored into variable EmpId, with the requirement that such an employee must be competent in the job category targeted by the application; *(iii)* *aState* evolves into state `received`. Formally:

$$\exists u{:}\mathsf{UserId}, j{:}\mathsf{JobCatId}, e{:}\mathsf{EmpId}, c{:}\mathsf{CompInId}\ \begin{pmatrix} pState = \texttt{enabled} \wedge aState = \texttt{undef} \\ \wedge\ u \neq \texttt{undef} \wedge j \neq \texttt{undef} \wedge e \neq \texttt{undef} \wedge c \neq \texttt{undef} \\ \wedge\ who(c) = e \wedge what(c) = j \\ \wedge\ pState' = \texttt{enabled} \wedge aState' = \texttt{received} \\ \wedge\ uId' = u \wedge jId' = j \wedge eId' = e \wedge cId' = c \end{pmatrix}$$

The second step transfers the application data into the application artifact relation, using its corresponding function variables, at the same resetting all application-related artifact variables to `undef` (including *aState*, so that new applications can be inserted). For the insertion, a "free" index (i.e., an index pointing to an undefined applicant) is picked. The newly inserted application gets a default score of `-1` (thus initializing it to "not eligible"), while the final result is `undef`:

$$\exists i{:}\mathsf{appIndex}\ \begin{pmatrix} pState = \texttt{enabled} \wedge aState = \texttt{received} \\ \wedge\ applicant[i] = \texttt{undef} \\ \wedge\ pState' = \texttt{enabled} \wedge aState' = \texttt{undef} \wedge cId' = \texttt{undef} \\ \wedge\ appJobCat' = \lambda j.\,(\mathsf{if}\ j = i\ \mathsf{then}\ jId\ \mathsf{else}\ appJobCat[j]) \\ \wedge\ applicant' = \lambda j.\,(\mathsf{if}\ j = i\ \mathsf{then}\ uId\ \mathsf{else}\ applicant[j]) \\ \wedge\ appResp' = \lambda j.\,(\mathsf{if}\ j = i\ \mathsf{then}\ eId\ \mathsf{else}\ appResp[j]) \\ \wedge\ appScore' = \lambda j.\,(\mathsf{if}\ j = i\ \mathsf{then}\ \texttt{-1}\ \mathsf{else}\ appScore[j]) \\ \wedge\ appResult' = \lambda j.\,(\mathsf{if}\ j = i\ \mathsf{then}\ \texttt{undef}\ \mathsf{else}\ appResult[j]) \\ \wedge\ jId' = \texttt{undef} \wedge uId' = \texttt{undef} \wedge eId' = \texttt{undef} \end{pmatrix}$$

Notice that such a transition does not prevent the possibility of inserting exactly the same application twice, at different indexes. If this is not wanted, the transition can be suitably changed so as to guarantee that no two identical applications can coexist in the same artifact relation (see Appendix A.1 for an example).

Each application currently considered as not eligible can be made eligible by assigning a proper score to it:

$$\exists i{:}\mathsf{appIndex}, s{:}\mathsf{Score}\ \begin{pmatrix} pState = \texttt{enabled} \wedge appScore[i] = \texttt{-1} \wedge s \geq 0 \\ \wedge\ pState' = \texttt{enabled} \wedge appScore'[i] = s \end{pmatrix}$$

Finally, application results are computed when the process moves to state *notified*. This is handled by the following *bulk* transition, which declares applications with a score above `80` as winning, and the others as losing:

$$pState = \texttt{enabled} \wedge pState' = \texttt{notified} \\ \wedge\ appResult' = \lambda j.\begin{pmatrix} \mathsf{if}\ appScore[j] > \texttt{80}\ \mathsf{then}\ \texttt{winner} \\ \mathsf{else}\ \texttt{loser} \end{pmatrix} \quad \triangleleft$$

As for SAS, a *safety* formula for $\mathcal{S}$ is a state formula $\upsilon(\underline{x})$. We say that $\mathcal{S}$ is *safe* with respect to υ if there is no DB-instance $\mathcal{M}$ of $\langle \Sigma_{ext}, T\rangle$, no $k \geq 0$, and no assignment in $\mathcal{M}$ to the variables $\underline{x}^0, \underline{a}^0 \ldots, \underline{x}^k, \underline{a}^k$ such that the formula

$$\iota(\underline{x}^0, \underline{a}^0) \wedge \tau(\underline{x}^0, \underline{a}^0, \underline{x}^1, \underline{a}^1) \\ \wedge \cdots \wedge \tau(\underline{x}^{k-1}, \underline{a}^{k-1}, \underline{x}^k, \underline{a}^k) \wedge \upsilon(\underline{x}^k, \underline{a}^k) \qquad (7)$$

is true in $\mathcal{M}$ (here $\underline{x}^i$, $\underline{a}^i$ are renamed copies of $\underline{x}$, $\underline{a}$). The safety problem is defined as for SAS.

Example 5. We consider a safety property for the RAS from Example 4 that checks whether, after having received the evaluation notification, the are no applicants left without winner or loser status being assigned:

$$\exists i{:}\mathsf{appIndex}\ \begin{pmatrix} pState = \texttt{notified} \wedge applicant[i] \neq \texttt{undef} \\ \wedge\ appResult[i] \neq \texttt{winner} \wedge appResult[i] \neq \texttt{loser} \end{pmatrix}$$

The job hiring RAS $\mathcal{S}_{hr}$ turns out to be safe w.r.t. this property (cf. Section 8). $\triangleleft$

Interestingly, we can still run backward search for handling safety problems in RASs. In analogy to Statement (1) of Theorem 1, we obtain:

Theorem 2. *Backward search (cf. Algorithm 1) is effective and partially correct for solving safety problems for RASs.* $\triangleleft$

Proof (sketch). We can keep Algorithm 1 almost unmodified. The procedure $\mathsf{QE}(T^*, \phi)$ mentioned on Line 6 can be extended so as to convert the preimage $Pre(\tau, \phi)$[13] of a state formula ϕ into a state formula (equivalent to it modulo the axioms of T^*): this is because a technical lemma ensures that T^* still eliminates from primitive formulae the existentially quantified variables over the basic sorts (elimination of quantified variables over artifact sorts is not possible, because these variables occur as arguments of artifact components). In addition, the satisfiability tests from Lines 2–3 can still be discharged (in fact, we prove that the entailment between state formulae can be decided via instantiation techniques). $\square$

Notice that the role of quantifier elimination (Line 6 of Algorithm 1) is twofold: *(i)* it allows to discharge the fixpoint test of Line 2 (see Lemma 4); *(ii)* it ensures termination in significant cases, namely those where *(strongly) local formulae* introduced below are involved.

7. TERMINATION RESULTS FOR RASs

[12] Notice that non-deterministic updates can be formalized using the existential quantified variables in the transition.

[13] Notice that in this case the definition of $Pre(\tau, \phi)$ gives us $\exists \underline{x}' \exists \underline{a}' (\tau(\underline{x}, \underline{a}, \underline{x}', \underline{a}') \wedge \phi(\underline{x}', \underline{a}'))$.

Theorem 2 gives a semi-decision procedure for unsafety: if the system is unsafe, the procedure discovers it, but if the system is safe, the procedure (still correct) may not terminate. Termination is much more difficult to achieve for RASs, since acyclicity of Σ is not sufficient to guarantee it. We present two termination results, both obtained via the use of well quasi-orders. I

7.1 Termination with Local Updates

Consider an acyclic signature Σ, a theory T (satisfying our Assumption 1), and an artifact setting $(\underline{x}, \underline{a})$ over an artifact extension Σ_{ext} of Σ. We call a state formula *local* if it is a disjunction of the formulae

$$\exists e_1 \cdots \exists e_k \left(\delta(e_1, \ldots, e_k) \wedge \textstyle\bigwedge_{i=1}^{k} \phi_i(e_i, \underline{x}, \underline{a}) \right), \qquad (8)$$

and *strongly local* if it is a disjunction of the formulae

$$\exists e_1 \cdots \exists e_n \left(\delta(e_1, \ldots, e_n) \wedge \psi(\underline{x}) \wedge \textstyle\bigwedge_{i=1}^{n} \phi_i(e_i, \underline{a}) \right). \qquad (9)$$

In (8) and (9), δ is a conjunction of variable equalities and inequalities, ϕ_i, ψ are quantifier-free, and $e_1, \ldots, e_n$ are individual variables varying over artifact sorts. The key expressivity limitation of local state formulae is that they cannot compare entries belonging to different tuples of artifact relations: in fact, each ϕ_i in (8) and (9) can contain only the existentially quantified variable e_i.

A transition formula $\hat{\tau}$ is *local* (resp., *strongly local*) if whenever a formula ϕ is local (resp., strongly local), so is $Pre(\hat{\tau}, \phi)$ (modulo the axioms of T^*). Examples of (strongly) local $\hat{\tau}$ are discussed in Appendix F.

THEOREM 3. *If Σ is acyclic, backward search (cf. Algorithm 1) terminates when applied to a local safety formula in a RAS whose τ is a disjunction of local transition formulae.* ◁

PROOF (SKETCH). Let $\tilde{\Sigma}$ be $\Sigma_{ext} \cup \{\underline{a}, \underline{x}\}$, that is, Σ_{ext} expanded with function symbols $\underline{a}$ and constants $\underline{x}$ (thus, a $\tilde{\Sigma}$-structure is a Σ_{ext}-structure endowed with an assignment to $\underline{x}$ and $\underline{a}$, which were variables and now are treated as symbols of $\tilde{\Sigma}$). We call a $\tilde{\Sigma}$-structure *cyclic*[14] if it is generated by one element belonging to the interpretation of an artifact sort. Since Σ is acyclic, so is $\tilde{\Sigma}$, and then one can show that there are only finitely many cyclic $\tilde{\Sigma}$-structures $\mathcal{C}_1, \ldots, \mathcal{C}_N$ up to isomorphism. With a $\tilde{\Sigma}$-structure $\mathcal{M}$ we associate the tuple of numbers $k_1(\mathcal{M}), \ldots, k_N(\mathcal{M}) \in \mathbb{N} \cup \{\infty\}$ counting the numbers of elements generating (as singletons) the cyclic substructures $\mathcal{C}_1, \ldots, \mathcal{C}_N$, respectively. Then we show that, if the tuple associated with $\mathcal{M}$ is componentwise bigger than the one associated with $\mathcal{N}$, then $\mathcal{M}$ satisfies all the local formulae satisfied by $\mathcal{N}$. Finally we apply Dikson Lemma [9]. □

Note that Theorem 3 can be used to subsume the decidability results of [37] concerning safety problems. Specifically, one needs to show that transitions in [37] are strongly local which, in turn, can be shown using quantifier elimination (see Appendix F for more details). Interestingly, Theorem 3 can be applied to more cases not covered in [37]. For example, one can provide transitions enforcing *updates over unboundedly many* tuples (bulk updates) that are strongly local (cf. Appendix F). One can also see that the safety problem for our running example is decidable since all its transitions are strongly local. Another case considers coverability problems for broadcast protocols [30, 25], which can be encoded using local formulae over the trivial one-sorted signature containing just one basic sort and finitely many constants (in the artifact setting, it is sufficient to take one artifact sort with one artifact component). These problems can be decided with a non-primitive recursive lower bound [42] (whereas the problems in [37] have an EXPSPACE upper bound).

[14] This is unrelated to cyclicity of Σ defined in Section 4, and comes from universal algebra terminology.

7.2 Termination with tree-like signatures

Σ is *tree-like* if it is acyclic and all non-leaf modes have outdegree 1. An artifact setting over Σ is tree-like if $\tilde{\Sigma} := \Sigma_{ext} \cup \{\underline{a}, \underline{x}\}$ is tree-like. In tree-like artifact settings, artifact relations have a single "data" component, and basic relations are unary or binary.

THEOREM 4. *Backward search (cf. Algorithm 1) terminates when applied to a safety problem in a RAS with a tree-like artifact setting.* ◁

PROOF (SKETCH). The crux of the proof is to show, using Kruskal's Tree Theorem [35], that the finitely generated $\tilde{\Sigma}$-structures are a well-quasi-order with respect to the embeddability partial order. □

Tree-like RAS are not subject to any locality restriction on their transitions. This allows to express sophisticated updates, including general bulk updates and transitions comparing different entries in artifact relations at once. The flight management process in Appendix A shows all these advanced features, with a tree-like RAS whose safety verification is indeed decidable.

8. FIRST EXPERIMENTS

We implemented a prototype of our backward reachability algorithm for artifact systems on top of the MCMT model checker. MCMT manages the verification in the infinite-state case by exploiting as its model-theoretic framework the declarative formalism of *array-based systems*. This formalism allows for symbolic representation of transitions and sets of states using formulae, whereas the system dynamics is defined by manipulating second order variables (i.e., arrays). Since their first introduction in [31, 32], array-based systems have been provided with various implementations of the standard backward reachability algorithms (including more sophisticated variants and heuristics). Starting from its first version [33], MCMT was successfully applied to cache coherence and mutual exclusions protocols [32], timed [19] and fault-tolerant [6, 5] distributed systems, and then to imperative programs [7, 8]; interesting case studies concerned waiting time bounds synthesis in parameterized timed networks [15] and internet protocols [14]. Further related tools include SAFARI [3] and ASASP [2]; finally,

CUBICLE [22] implements the array-based setting on a parallel architecture with further powerful extensions.

The MCMT work principle is rather simple: the tool generates the proof obligations arising from the safety and fixpoint tests in backward search (lines 2-3 of Algorithm 1) and passes them to the background SMT-solver (currently it is YICES [29]). In practice, the situation is more complicated because SMT-solvers are quite efficient in handling satisfiability problems in combined theories at quantifier-free level, but may encounter difficulties with quantifiers. For this reason, MCMT implements modules for *quantifier elimination* and *quantifier instantiation*. A *specific module* for the quantifier elimination problems mentioned in line 6 of Algorithm 1 has been added to version 2.8 of MCMT.

We produced a benchmark consisting of eight realistic business process examples and ran it in MCMT (detailed explanations and results are given in Appendix G). The examples are partially made by hand and partially obtained from those supplied in [37]. A thorough comparison with VERIFAS [37] is at the moment rather problematic, for the reasons mentioned below. First, we deal with safety problems, whereas VERIFAS handles general LTL-FO properties. At the same time, our setting is more expressive (for instance, we cover "bulk" updates). Second, MCMT is based on backward reachability, while VERIFAS employs forward search. Hence, the state space representation in the two approaches may differ extensively depending on the given system and property. Third, MCMT allows the user to specify custom safety properties, while VERIFAS works on predefined LTL-FO templates, instantiated based on syntactic criteria (not defined by the user). Finally, the tools use different input specification languages.

The benchmark is available as part of the last distribution 2.8 of the tool.[15] Table 1 shows the very encouraging results (the first row tackles Example 5). While a systematic evaluation is matter of future work, MCMT seems to effectively handle the benchmark with a similar performance to that shown in other, well-established settings, with verification times below 1s in most cases.

Exp.	#(AC)	#(AV)	#(T)	Property	Result	Time (s)
E1	9	18	15	E1P1	SAFE	0.06
				E1P2	UNSAFE	0.36
				E1P3	UNSAFE	0.50
				E1P4	UNSAFE	0.35
E2	6	13	28	E2P1	SAFE	0.72
				E2P2	UNSAFE	0.88
				E2P3	UNSAFE	1.01
				E2P4	UNSAFE	0.83
E3	4	14	13	E3P1	SAFE	0.05
				E3P2	UNSAFE	0.06
E4	9	11	21	E4P1	SAFE	0.12
				E4P2	UNSAFE	0.13
E5	6	17	34	E5P1	SAFE	4.11
				E5P2	UNSAFE	0.17
E6	2	7	15	E6P1	SAFE	0.04
				E6P2	UNSAFE	0.08
E7	2	28	38	E7P1	SAFE	1.00
				E7P2	UNSAFE	0.20
E8	3	20	19	E8P1	SAFE	0.70
				E8P2	UNSAFE	0.15

Table 1: Experimental results. The size of the input system is reflected by columns #(AC), #(AV), #(T), indicating the number of artifact components, artifact variables, and transitions.

9. CONCLUSION

We have laid the foundations of SMT-based verification for artifact systems, focusing on safety problems and relying on array-based systems as underlying formal model. We have shown how to overcome the main technical difficulty arising from this approach, namely reconstructing quantifier elimination techniques in the rich setting of artifact systems, using the model-theoretic machinery of model completion. We have then exploited the so-obtained framework to homogeneously reconstruct and extend known results on the decidability of verification of artifact systems, and to single out a novel, decidable class. The presented techniques have been implemented on top of the well-established MCMT model checker, making our approach fully operational.

From the foundational point of view, we plan to use the present contribution as the starting point for a full line of research dedicated to SMT-based techniques for the effective verification of data-aware processes, considering richer forms of verification going beyond safety, and richer classes of artifact systems incorporating concrete data types and arithmetic operations.

From the practical point of view, we intend to start from the encouraging results reported here, and account for an extensive experimental evaluation of our approach, using the VERIFAS system as a baseline. A natural next step is then to study how well-established techniques for SMT-based model checking can be used to speed up the verification of artifact systems.

Finally, we plan to tackle more conventional process modeling notations, in particular data-aware extensions of the de-facto standard BPMN.

[15] `http://users.mat.unimi.it/users/ghilardi/mcmt/`, subdirectory `/examples/dbdriven` of the distribution. The user manual contains a new section giving essential information on how to produce user-defined examples.

10. REFERENCES

[1] P. A. Abdulla, C. Aiswarya, M. F.and M. Montali Atig, and O. Rezine. Recency-bounded verification of dynamic database-driven systems. In *Proc. PODS*, 2016.

[2] F. Alberti, A. Armando, and S. Ranise. ASASP: automated symbolic analysis of security policies. In *Proc. CADE*, 2011.

[3] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *Proc. CAV*, 2012.

[4] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Form. Methods Syst. Des.*, 45(1), 2014.

[5] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Brief announcement: Automated support for the design and validation of fault tolerant parameterized systems - A case study. In *Proc. DISC*, 2010.

[6] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, 8(1/2), 2012.

[7] F. Alberti, S. Ghilardi, and N. Sharygina. Booster: An acceleration-based verification framework for array programs. In *Proc. ATVA*, 2014.

[8] F. Alberti, S. Ghilardi, and N. Sharygina. A framework for the verification of parameterized infinite-state systems. *Fund. Inform.*, 150(1), 2017.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *Proc. PODS*, 2013.

[11] F. Belardinelli, A. Lomuscio, and F. Patrizi. An abstraction technique for the verification of artifact-centric systems. In *Proc. KR*, 2012.

[12] M. Bojańczyk, L. Segoufin, and S. Toruńczyk. Verification of database-driven systems via amalgamation. In *Proc. PODS*, 2013.

[13] A. R. Bradley and Z. Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[14] D. Bruschi, A. Di Pasquale, S. Ghilardi, A. Lanzi, and E. Pagani. Formal verification of ARP (address resolution protocol) through SMT-based model checking - A case study. In *Proc. IFM*, 2017.

[15] R. Bruttomesso, A. Carioni, S. Ghilardi, and S. Ranise. Automated analysis of parametric timing-based mutual exclusion algorithms. In *Proc. NFM*, 2012.

[16] D. Calvanese, G. De Giacomo, and M. Montali. Foundations of data aware process analysis: A database theory perspective. In *Proc. PODS*, 2013.

[17] D. Calvanese, G. De Giacomo, M. Montali, and F. Patrizi. First-order mu-calculus over generic transition systems and applications to the situation calculus. *Inf. and Comp.*, 2017.

[18] D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin. Quantifier elimination for database driven verification. Technical Report arXiv:1806.09686, arXiv.org, 2018.

[19] A. Carioni, S. Ghilardi, and S. Ranise. MCMT in the land of parametrized timed automata. In *Proc. VERIFY*, 2010.

[20] A. Carioni, S. Ghilardi, and S. Ranise. Automated termination in model-checking modulo theories. *Int. J. Found. Comput. Sci.*, 24(2), 2013.

[21] C.-C. Chang and J. H. Keisler. *Model Theory*. North-Holland Publishing Co., 1990.

[22] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle: A parallel SMT-based model checker for parameterized systems - Tool paper. In *Proc. CAV*, 2012.

[23] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM TODS*, 37(3), 2012.

[24] E. Damaggio, R. Hull, and R. Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. In *Proc. BPM*, 2011.

[25] G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. In *Proc. CSL*, 1999.

[26] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *Proc. ICDT*, 2009.

[27] A. Deutsch, Y. Li, and V. Vianu. Verification of hierarchical artifact systems. In *Proc. PODS*, 2016.

[28] M. Dumas. On the convergence of data and process engineering. In *Proc. ADBIS*, 2011.

[29] B. Dutertre and L. De Moura. The YICES SMT solver. Technical report, SRI International, 2006.

[30] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS*, 1999.

[31] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model checking of array-based systems. In *Proc. IJCAR*, 2008.

[32] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Log. Methods Comput. Sci.*, 6(4), 2010.

[33] S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *Proc. IJCAR*, 2010.

[34] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *Proc. OTM*, 2008.

[35] J. B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.*, 95, 1960.

[36] V. Künzle, B. Weber, and M Reichert. Object-aware business processes: Fundamental requirements and their support in existing approaches. *Int. J. of Information System*

Modeling and Design, 2(2), 2011.

[37] Y. Li, A. Deutsch, and V. Vianu. VERIFAS: A practical verifier for artifact systems. *PVLDB*, 11(3), 2017.

[38] A. Meyer, S. Smirnov, and M. Weske. Data in business processes. Technical Report 50, Hasso-Plattner-Institut for IT Systems Engineering, Universität Potsdam, 2011.

[39] M. Reichert. Process and data: Two sides of the same coin? In *Proc. OTM*, 2012.

[40] C. Richardson. Warning: Don't assume your business processes use master data. In *Proc. BPM*, 2010.

[41] A. Robinson. *On the metamathematics of algebra*. North-Holland Publishing Co., 1951.

[42] S. Schmitz and P. Schnoebelen. The power of well-structured systems. In *Proc. CONCUR*, 2013.

[43] Bruce Silver. *BPMN Method and Style*. Cody-Cassidy, 2nd edition, 2011.

[44] V. Vianu. Automatic verification of database-driven systems: a new frontier. In *Proc. ICDT*, 2009.

[45] William H. Wheeler. Model-companions and definability in existentially complete structures. *Israel J. Math.*, 25(3-4), 1976.

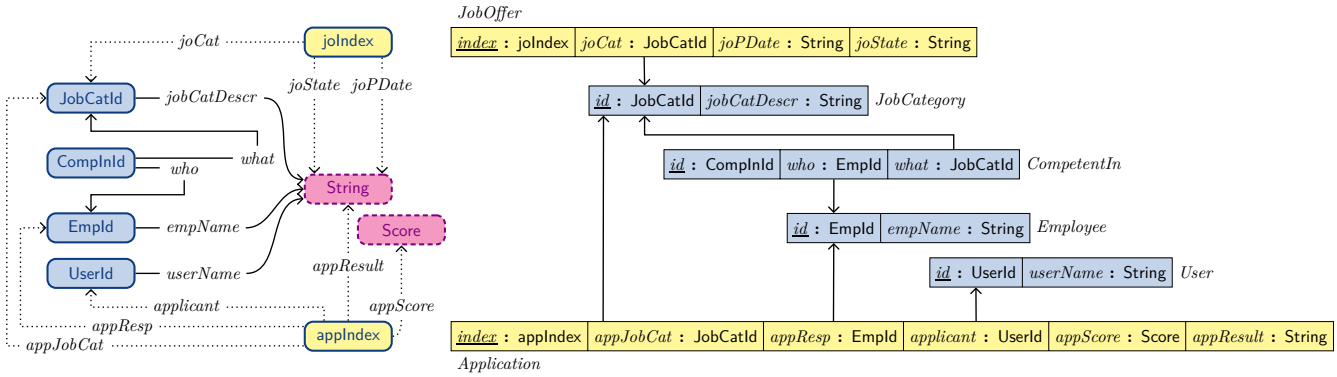


Figure 2: On the left: characteristic graph of the human resources DB signature from Example 1, augmented with the signature of the artifact extension for the job hiring process; value sorts are shown in pink, basic id sorts in blue, and artifact id sorts in yellow. On the right: relational view of the DB signature and the corresponding artifact relations; each cell denotes an attribute with its type, underlined attributes denote primary keys, and directed edges capture foreign keys.

APPENDIX

A. EXAMPLES

In this section, we present two full examples of RAS for which our backward reachability technique terminates. In particular, they are meant to highlight the expressiveness of our approach, even in presence of the restrictions imposed by Theorems 3 and 4 towards decidability of reachability. When writing transition formulae in the examples, we make the following assumption: when an artifact variable or component is not mentioned at all in a transition, it is meant that is updated identically; if it is mentioned, the relevant update function in the transition will specify how it is updated.[16]

A.1 Job Hiring Process

We present a RAS $\mathcal{S}_{hr}$ capturing a job hiring process where multiple job categories may be turned into actual job offers, each one receiving many applications from registered users. Such applications are then evaluated, finally deciding which are accepted and which are rejected. The example is inspired by the job hiring process presented in [43] to show the intrinsic difficulties of capturing real-life processes with many-to-many interacting business entities using conventional process modeling notations (such as BPMN).

As for the read-only DB, $\mathcal{S}_{hr}$ works over the DB schema of Example 1, extended with a further value sort Score used to score the applications sent for job offerings. Score contains 102 different values, intuitively corresponding to the integer numbers from -1 to 100 (included), where -1 denotes that the application is considered to be not eligible, while a score between 0 and 100 indicates the actual score assigned after evaluating the application. For the sake of readability, we make use of the usual integer comparison predicates to compare variables of type Score. This is simply syntactic sugar and does not require the introduction of rigid predicates in our framework. In fact, given two variables x and y of type Score, $x < y$ is a shortcut for the finitary disjunction testing that x is one of the scores that are "less than" y (similarly for the other comparison predicates).

As for the working memory, $\mathcal{S}_{hr}$ consists of three artifacts: a single-instance *job hiring* artifact tracking the three main phases of the overall process, and two multi-instance artifacts accounting for the evolution of *job offers*, and that of corresponding *user applications*. The job hiring artifact simply requires a dedicated *pState* variable to store the current process state. The job offer and user application multi-instance artifacts are instead modeled by enriching the DB signature Σ_{hr} of the read-only database of human resources. In particular, an artifact extension is added containing two artifact sorts joIndex and appIndex used to respectively *index* (i.e., *"internally" identify*) job offers and applications. The management of job offers and applications is then modeled by a full-fledged artifact setting that adopts:

- artifact components with domains joIndex and appIndex to capture the artifact relations storing multiple instances of job offers and applications;
- individual variables used as temporary memory to manipulate the artifact relations.

The actual components of such an artifact setting will be introduced when needed.

We now describe how the process works, step by step. Initially, hiring is disabled, which is captured by initially setting the *pState* variable to `undef`. A transition of the process from disabled to *enabled* may occur provided that

[16]Notice that non-deterministic updates can be formalized using the existential quantified variables in the transition.

the read-only HR DB contains at least one registered user (who, in turn, may decide to apply for job offers created during this phase). Technically, we introduce a dedicated artifact variable *uId* initialized to `undef`, and used to load the identifier of such a registered user, if (s)he exists. The enablement task is then captured by the following transition formula:

$$\exists y : \mathsf{UserId}\ \big(pState = \texttt{undef} \wedge y \neq \texttt{undef} \wedge pState' = \texttt{enabled} \wedge uId' = y\big)$$

We now focus on the creation of a job offer. When the overall hiring process is enabled, some job categories present in the read-only DB may be published into a corresponding job offer, consequently becoming ready to receive applications. This is done in two steps. In the first step, we transfer the id of the job category to be published to the artifact variable *jId*, and the string representing the publishing date to the artifact variable *pubDate*. Thus, *jId* is filled with the identifier of a job category picked from JobCatId (modeling a nondeterministic choice of category), while *pubDate* is filled with a String (modeling a *user input* where one of the infinitely many strings is injected into *pubDate*).

In addition, the transition interacts with a further artifact variable *pubState* capturing the publishing state of offers, and consequently used to synchronize the two steps for publishing a job offer. In particular, this first step can be executed only if *pubState* is *not* in state `publishing`, and has the effect of setting it to such a value, thus preventing the first step to be executed twice in a row (which would actually overwrite what has been stored in *jId* and *pubDate*). Technically, we have:

$$\exists j{:}\mathsf{JobCatId}, d{:}\mathsf{String}\ \begin{pmatrix} pState = \texttt{enabled} \wedge pubState \neq \texttt{publishing} \wedge j \neq \texttt{undef} \\ \wedge\ pState' = \texttt{enabled} \wedge pubState' = \texttt{publishing} \wedge jId' = j \wedge pubDate' = d \end{pmatrix}$$

The second step consists in transferring the content of these three variables into corresponding artifact components that keep track of all active job offers, at the same time resetting the content of the artifact variables to `undef`. This is done by introducing three function variables with domain JoIndex, respectively keeping track of the category, publishing date, and state of job offers:

$$\begin{array}{lcl} joCat & : \mathsf{JoIndex} \longrightarrow \mathsf{JobCatId} \\ joPDate & : \mathsf{JoIndex} \longrightarrow \mathsf{String} \\ joState & : \mathsf{JoIndex} \longrightarrow \mathsf{String} \end{array}$$

With these artifact components at hand, the second step is then realized as follows:

$$\exists i{:}\mathsf{JoIndex}$$
$$\left(\begin{array}{l} pState = \texttt{enabled} \wedge pubState = \texttt{publishing} \wedge joPDate[i] = \texttt{undef} \wedge joCat[i] = \texttt{undef} \wedge joState[i] = \texttt{undef} \\ \wedge\ aState' = \texttt{undef} \wedge pState' = \texttt{enabled} \wedge pubState' = \texttt{published} \\ \wedge\ joCat' = \lambda j.\ \begin{pmatrix} \mathsf{if}\ j = i\ \mathsf{then}\ jId \\ \mathsf{else\ if}\ joCat[j] = jId\ \mathsf{then}\ \texttt{undef} \\ \mathsf{else}\ joCat[j] \end{pmatrix} \wedge joPDate' = \lambda j.\ \begin{pmatrix} \mathsf{if}\ j = i\ \mathsf{then}\ pubDate \\ \mathsf{else\ if}\ joCat[j] = jId\ \mathsf{then}\ \texttt{undef} \\ \mathsf{else}\ joPDate[j] \end{pmatrix} \\ \wedge\ joState' = \lambda j.\ \begin{pmatrix} \mathsf{if}\ j = i\ \mathsf{then}\ \texttt{open} \\ \mathsf{else\ if}\ joCat[j] = jId\ \mathsf{then}\ \texttt{undef} \\ \mathsf{else}\ joState[j] \end{pmatrix} \\ \wedge\ uId' = \texttt{undef} \wedge eId' = \texttt{undef} \wedge jId' = \texttt{undef} \wedge pubDate' = \texttt{undef} \wedge cId' = \texttt{undef} \end{array}\right)$$

The "if-then-else" pattern is used to create an entry for the job offer artifact relation containing the information stored into the artifact variables populated in the first step, at the same time *making sure that only one entry exists for a given job category*. This is done by picking a job offer index i that is not already pointing to an actual job offer, i.e., such that the i-th element of *joCat* is `undef`. Then, the transition updates the whole content of the three artifact components *joCat*, *joPDate*, and *joState* as follows:

- The i-th entry of such variables is respectively assigned to the job category stored in JobCatId, the string stored in *pubDate*, and the constant `open` (signifying that this entry is ready to receive applications).
- All other entries are kept unaltered, with the exception of a possibly existing entry j with $j \neq i$ that points to the same job category contained in JobCatId. If such an entry j exists, its content is reset, by assigning to the j-th component of all three artifact components the value `undef`. Obviously, other strategies to resolve this possible conflict can be seamlessly captured in our framework.

A similar conflict resolution strategy will be used in the other transitions of this example.

We now focus on the evolution of applications to job offers. Each application consists of a job category, the identifier of the applicant user, the identifier of an employee from human resources who is responsible for the application, the score assigned to the application, and the application final result (indicating whether the application is among the winners or the losers for the job offer). These five information types are encapsulated into five dedicated function

variables with domain $\mathsf{appIndex}$, collectively realizing the application artifact relation:

$$\begin{array}{lcl} \mathit{appJobCat} & : \mathsf{appIndex} \longrightarrow \mathsf{JobCatId} \\ \mathit{applicant} & : \mathsf{appIndex} \longrightarrow \mathsf{UserId} \\ \mathit{appResp} & : \mathsf{appIndex} \longrightarrow \mathsf{EmpId} \\ \mathit{appScore} & : \mathsf{appIndex} \longrightarrow \mathsf{Score} \\ \mathit{appResult} & : \mathsf{appIndex} \longrightarrow \mathsf{String} \end{array}$$

With these function variables at hand, we discuss the insertion of an application into the system for an open job offer. This is again managed in multiple steps, first loading the necessary information into dedicated artifact variables, and finally transferring them into the function variables that collectively realize the application artifact relation. To synchronize these multiple steps and define which step is applicable in a given state, we make use of a string artifact variable called *aState*. The first step to insert an application is executed when *aState* is `undef`, and has the effect of loading into *jId* the identifier of a job category that has a corresponding open job offer, at the same time putting *aState* in state `joSelected`.

$$\exists i{:}\mathsf{joIndex} \left(\begin{array}{l} \mathit{pState} = \texttt{enabled} \wedge \mathit{aState} = \texttt{undef} \wedge \mathit{joCat}[i] \neq \texttt{undef} \wedge \mathit{joState}[i] = \texttt{open} \\ \wedge\ \mathit{pState}' = \texttt{enabled} \wedge \mathit{aState}' = \texttt{joSelected} \wedge \mathit{jId}' = \mathit{joCat}[i] \wedge \mathit{joCat}' = \mathit{joCat} \\ \wedge\ \mathit{uId}' = \texttt{undef} \wedge \mathit{eId}' = \texttt{undef} \wedge \mathit{jId}' = \texttt{undef} \wedge \mathit{pubDate}' = \texttt{undef} \wedge \mathit{cId}' = \texttt{undef} \end{array} \right)$$

The last row of the transition resets the content of all artifact variables, cleaning the working memory for the forthcoming steps (avoiding that stale values are present there). This is also useful from the technical point of view, as it guarantees that the transition is *strongly local* (cf. Section 7.1, and the discussion in Appendix F.1).

The second step has a twofold purpose: picking the identifier of the user who wants to submit an application for the selected job offer, and assigning to its application an employee of human resources who is competent in the category of the job offer. This also results in an update of variable *aState*:

$$\exists u{:}\mathsf{UserId}, e{:}\mathsf{EmpId}, c{:}\mathsf{CompInId} \left(\begin{array}{l} \mathit{pState} = \texttt{enabled} \wedge \mathit{aState} = \texttt{joSelected} \wedge \mathit{who}(c) = e \wedge \mathit{what}(c) = \mathit{jId} \wedge \mathit{jId} \neq \texttt{undef} \wedge u \neq \texttt{undef} \wedge c \neq \texttt{undef} \\ \wedge\ \mathit{pState}' = \texttt{enabled} \wedge \mathit{aState}' = \texttt{received} \wedge \mathit{jId}' = \mathit{jId} \wedge \mathit{uId}' = u \wedge \mathit{eId}' = e \wedge \mathit{cId}' = c \end{array} \right)$$

The last step transfers the application data into the application artifact relation, making sure that no two applications exist for the same user and the same job category. The transfer is done by assigning the artifact variables to corresponding components of the application artifact relation, at the same resetting all application-related artifact variables to `undef` (including *aState*, so that new applications can be inserted). For the insertion, a "free" index (i.e., an index pointing to an undefined applicant, with an undefined job category and an undefined responsible) is picked. The newly inserted application gets a default score of `-1` (thus initializing it to "not eligible"), while the final result is `undef`:

$$\exists i{:}\mathsf{appIndex} \left(\begin{array}{l} \mathit{pState} = \texttt{enabled} \wedge \mathit{aState} = \texttt{received} \\ \wedge\ \mathit{appJobCat}[i] = \texttt{undef} \wedge \mathit{applicant}[i] = \texttt{undef} \wedge \mathit{appResp}[i] = \texttt{undef} \\ \wedge \mathit{pState}' = \texttt{enabled} \wedge \mathit{aState}' = \texttt{undef} \\ \wedge\ \mathit{appJobCat}' = \lambda j. \left(\begin{array}{l} \mathsf{if}\ j = i\ \mathsf{then}\ \mathit{jId} \\ \mathsf{else\ if}\ \ (\mathit{applicant}[j] = \mathit{uId} \wedge \mathit{appResp}[j] = \mathit{eId})\ \ \mathsf{then}\ \texttt{undef} \\ \qquad \mathsf{else}\ \mathit{appJobCat}[j] \end{array} \right) \\ \wedge\ \mathit{applicant}' = \lambda j. \left(\begin{array}{l} \mathsf{if}\ j = i\ \mathsf{then}\ \mathit{uId} \\ \mathsf{else\ if}\ \ (\mathit{applicant}[j] = \mathit{uId} \wedge \mathit{appResp}[j] = \mathit{eId})\ \ \mathsf{then}\ \texttt{undef} \\ \qquad \mathsf{else}\ \mathit{applicant}[j] \end{array} \right) \\ \wedge\ \mathit{appResp}' = \lambda j. \left(\begin{array}{l} \mathsf{if}\ j = i\ \mathsf{then}\ \mathit{eId} \\ \mathsf{else\ if}\ \ (\mathit{applicant}[j] = \mathit{uId} \wedge \mathit{appResp}[j] = \mathit{eId})\ \ \mathsf{then}\ \texttt{undef} \\ \qquad \mathsf{else}\ \mathit{appResp}[j] \end{array} \right) \\ \wedge\ \mathit{appScore}' = \lambda j. \left(\begin{array}{l} \mathsf{if}\ j = i\ \mathsf{then}\ \texttt{-1} \\ \mathsf{else\ if}\ \ (\mathit{applicant}[j] = \mathit{uId} \wedge \mathit{appResp}[j] = \mathit{eId})\ \ \mathsf{then}\ \texttt{undef} \\ \qquad \mathsf{else}\ \mathit{appScore}[j] \end{array} \right) \\ \wedge\ \mathit{appResult}' = \lambda j. \left(\begin{array}{l} \mathsf{if}\ j = i \vee (\mathit{applicant}[j] = \mathit{uId} \wedge \mathit{appResp}[j] = \mathit{eId})\ \ \mathsf{then}\ \texttt{undef} \\ \mathsf{else}\ \mathit{appResult}[j] \end{array} \right) \\ \wedge\ \mathit{uId}' = \texttt{undef} \wedge \mathit{eId}' = \texttt{undef} \wedge \mathit{jId}' = \texttt{undef} \wedge \mathit{pubDate}' = \texttt{undef} \wedge \mathit{cId}' = \texttt{undef} \end{array} \right)$$

Each single application that is currently considered as not eligible can be made eligible by carrying out an evaluation that assigns a proper score to it. This is managed by the following transition:

$$\exists i{:}\mathsf{appIndex}, s{:}\mathsf{Score} \left(\begin{array}{l} \mathit{pState} = \texttt{enabled} \wedge \mathit{applicant}[i] \neq \texttt{undef} \wedge \mathit{appScore}[i] = \texttt{-1} \wedge s \geq 0 \\ \wedge\ \mathit{pState}' = \texttt{enabled} \wedge \mathit{appScore}'[i] = s \end{array} \right)$$

Evaluations are only possible as long as the process is in the `enabled` state. The process moves from enabled to *final* once the deadline for receiving applications to job offers is actually reached. This event is captured with pure nondeterminism, and has the additional *bulk* effect of turning all open job offers to *closed*:

$$pState = \texttt{enabled} \wedge pState' = \texttt{final} \wedge joState' = \lambda j. \begin{pmatrix} \text{if } joState[j] = \texttt{open} \text{ then } \texttt{closed} \\ \text{else } joState[j] \end{pmatrix}$$

Finally, we consider the determination of winners and losers, which is carried out when the overall hiring process moves from final to *notified*. This is captured by the following *bulk* transition, which declares all applications with a score above 80 as winning, and all the others as losing:

$$pState = \texttt{final} \wedge pState' = \texttt{notified} \wedge appResult' = \lambda j. \begin{pmatrix} \text{if } appScore[j] > 80 \text{ then } \texttt{winner} \\ \text{else } \texttt{loser} \end{pmatrix}$$

We close the example with the following key observation. All transitions of the hiring process are, in their current form, strongly local, with the exception of those operating over artifact relations in a way that ensures no repeated entries are inserted. Such transitions can be turned into strongly local ones if *repetitions in the artifact relations are allowed*. That is, multiple identical job offers and applications can be inserted in the corresponding relations, using different indexes. This is the strategy adopted in Example 4 in the main text of the paper. This approach realizes a sort of multiset semantics for artifact relations. The impact of this variant to verification of safety properties is discussed in Appendix F.2.

A.2 Flight Management Process

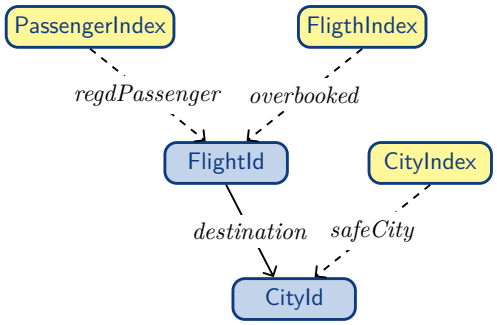


Figure 3: A characteristic graph of the flight management process, where blue and yellow boxes respectively represent basic and artifact sorts.

In this section we consider a simple RAS that falls in the scope of the decidability result described in Section 7. Specifically, this example has a tree-like artifact setting (see Figure 3), thus assuring that, when solving the safety problem for it, the backward search algorithm is guaranteed to terminate. Note, however, that the termination result adopted here is the one of Theorem 4 due to the non-locality of certain transitions, as explained in detail below.

The flight management process represents a simplified version of a flight management system adopted by an airline. To prepare a flight, the company picks a corresponding destination (that meets the aviation safety compliance indications) and consequently reports on a number of passengers that are going to attend the flight. Then, an airport dispatcher may pick a manned flight and put it in the airports flight plan. In case the flight destination becomes unsafe (e.g., it was stroke by a hurricane or the hosting airport had been seized by terrorists), the dispatcher uses the system to inform the airline about this condition. In turn, the airline notifies all the passengers of the affected destination about the contingency, and temporary cancels their flights.

To formalize these different aspects, we make use of a DB signature Σ_{fm} that consists of: *(i)* two id sorts, used to identify flights and cities; *(ii)* one function symbol $destination : \mathsf{FlightId} \longrightarrow \mathsf{CityId}$ mapping flight identifiers to their corresponding destinations (i.e., city identifiers). Note that, in a classical relational model (cf. Section 4.1), our signature would contain two relations: one binary R_{FlightId} that defines flights and their destinations, and another unary R_{CityId} identifying cities, that are referenced by R_{FlightId} using *destination*.

We assume that the read-only flight management database contains data about at least one flight and one city. To start the process, one needs at least one city to meet the aviation safety compliances. It is assumed that, initially, all the cities are unsafe. An airport dispatcher, at once, may change the safety status only of one city.

We model this action by performing two consequent actions. First, we select the city identifier and store it in the designated artifact variable *safeCitytId*:

$$\exists c{:}\mathsf{CityId}\,(c \neq \texttt{undef} \wedge safeCitytId = \texttt{undef} \wedge safeCitytId' = c)$$

Then, we place the extracted city identifier into a unary artifact relation $safeCity : \mathsf{CityIndex} \longrightarrow \mathsf{CityId}$, that is used

to represent safe cities and where CityIndex is its artifact sort.

$$\exists i{:}\mathsf{CityIndex}\ \left(\begin{array}{l} safeCity[i] = \mathtt{undef} \wedge safeCitytId \neq \mathtt{undef} \wedge safeCitytId' = \mathtt{undef} \\ \wedge\ safeCity' = \lambda j.\ \left(\begin{array}{l}\mathsf{if}\ j = i\ \mathsf{then}\ safeCitytId \\ \mathsf{else\ if}\ safeCity[j] = safeCitytId\ \mathsf{then}\ \mathtt{undef} \\ \quad \mathsf{else}\ safeCity[j]\end{array}\right)\end{array}\right)$$

Note that two previous transitions can be rewritten as a unique one, hence showing a more compact way of specifying RAS transitions. This, in turn, can augment the performance of the verifier while working with large-scale cases. The unified transition actually looks as follows:

$$\exists c{:}\mathsf{CityId}, \exists i{:}\mathsf{CityIndex}\ \left(\begin{array}{l} c \neq \mathtt{undef} \wedge safeCity[i] = \mathtt{undef} \\ \wedge\ safeCity' = \lambda j.\ \left(\begin{array}{l}\mathsf{if}\ j = i\ \mathsf{then}\ c \\ \mathsf{else\ if}\ safeCity[j] = c\ \mathsf{then}\ \mathtt{undef} \\ \quad \mathsf{else}\ safeCity[j]\end{array}\right)\end{array}\right)$$

Then, to register passengers with booked tickets on a flight, the airline needs to make sure that a corresponding flight destination is actually safe. To perform the passenger registration, the airline selects a flight identifier that is assigned to the route and uses it to populate entries in an unary artifact relation $regdPassenger : \mathsf{PassengerIndex} \longrightarrow \mathsf{FlightId}$. Note that there may be more than one passenger taking the flight, and therefore, more than one entry in $regdPassenger$ with the same flight identifier.

$$\exists i{:}\mathsf{CityIndex}, f{:}\mathsf{FlightId}, p{:}\mathsf{PassengerIndex}\ \left(\begin{array}{l} f \neq \mathtt{undef} \wedge destination(f) = safeCity[i] \wedge regdPassenger[p] = \mathtt{undef} \\ \wedge\ regdPassenger' = \lambda j.\ \left(\begin{array}{l}\mathsf{if}\ j = p\ \mathsf{then}\ f \\ \mathsf{else}\ regdPassenger[j]\end{array}\right)\end{array}\right)$$

We also assume that the airline owns aircraft of one type that can contain no more than k passengers. In case there were more than k passengers registered on the flight, the airline receives a notification about its overbooking and temporary suspends all passenger registrations associated to this flight. This is modelled by checking whether there are at least $k + 1$ entries in $regdPassenger$. If so, the flight identifier is added to a unary artifact relation $overbooked : \mathsf{FligthIndex} \longrightarrow \mathsf{FlightId}$ and all the passenger registrations in $regdPassenger$ that reference this flight identifier are nullified by updating unboundedly many entries in the corresponding artifact relation:[17]

$$\exists p_1{:}\mathsf{PassengerIndex}, \ldots p_{k+1}{:}\mathsf{PassengerIndex}, m{:}\mathsf{FligthIndex}\ \left(\begin{array}{l} \bigwedge_{i,i' \in \{1,\ldots,k+1\}, i \neq i'} (p_i \neq p_{i'} \wedge regdPassenger[p_i] \neq \mathtt{undef} \wedge regdPassenger[p_i] = regdPassenger[p_{i'}]) \\ \wedge\ overbooked[m] = \mathtt{undef} \\ \wedge\ regdPassenger' = \lambda j.\ \left(\begin{array}{l}\mathsf{if}\ regdPassenger[j] = regdPassenger[p_1]\ \mathsf{then}\ \mathtt{undef} \\ \mathsf{else}\ regdPassenger[j]\end{array}\right) \\ \wedge\ overbooked'[m] = regdPassenger[p_1]\end{array}\right)$$

Notice that this transition is not local, since its guard contains literals of the form $regdPassenger[p_i] = regdPassenger[p_{i'}]$ (with $p_i \neq p_{i'}$), which involve more than one element of one artifact sort.

In case of any contingency, the airport dispatcher may change the city status from *safe* to *unsafe*. To do it, we first select one of the safe cities, make it unsafe (i.e., remove it from *safeCity* relation) and store its identifier in the artifact variable *unsafeCityId*:

$$\exists i{:}\mathsf{CityIndex}\ \big(unsafeCityId = \mathtt{undef} \wedge safeCity[i] \neq \mathtt{undef} \wedge unsafeCityId' = safeCity[i] \wedge safeCity'[i] = \mathtt{undef}\big)$$

Then, we use the remembered city identifier to cancel all the passenger registrations for flights that use this city as their destination:[18]

$$\left(\begin{array}{l} unsafeCityId \neq \mathtt{undef} \wedge unsafeCityId' = \mathtt{undef} \\ \wedge\ regdPassenger' = \lambda j.\ \left(\begin{array}{l}\mathsf{if}\ destination(regdPassenger[j]) = unsafeCityId\ \mathsf{then}\ \mathtt{undef} \\ \mathsf{else}\ regdPassenger[j]\end{array}\right)\end{array}\right)$$

Also in this case, we can shrink the transitions into a single transition:

$$\exists i{:}\mathsf{CityIndex}\left(safeCity[i] \neq \mathtt{undef} \wedge regdPassenger' = \lambda j.\ \left(\begin{array}{l}\mathsf{if}\ destination(regdPassenger[j]) = safeCity[i]\ \mathsf{then}\ \mathtt{undef} \\ \mathsf{else}\ regdPassenger[j]\end{array}\right)\right)$$

However, as in the previous case, the transition turns out to be not local. Specifically, it is due to the literal

[17] For simplicity of presentation, we simply remove such data from the artifact relation. In a real setting, this information would actually be transferred to a dedicated, historical table, so as to reconstruct the status of past, overbooked flights.

[18] Similarly to the previous case, the corresponding transition performs the intended action by updating unboundedly many entries in the artifact relation.

$destination(regdPassenger[j]) = safeCity[i]$ that involves more than one element with different artifact sorts.

B. PROOFS AND COMPLEMENTS FOR SECTION 4

We fix a signature Σ and a universal theory T as in Definition 1.

Observe that if Σ is acyclic, there are only finitely many terms involving a single variable x: in fact, there are as many terms as paths in $G(\Sigma)$ starting from the sort of x. If k_Σ is the maximum number of terms involving a single variable, then (since all function symbols are unary) there are at most k_Σ^n terms involving n variables.

Proposition 1. *T has the finite model property in case Σ is acyclic.*

PROOF. If $T := \emptyset$, then congruence closure ensures that the finite model property holds and decides constraint satisfiability in time $O(n \log n)$ [13].

Otherwise, we reduce the argument to the Herbrand Theorem. Indeed, suppose to have a set Φ of universal formulae. Herbrand Theorem states that Φ has a model iff the set of ground instances of Φ has a model. These ground instances are finitely many by acyclicity, so we can reduce to the case where T is empty. □

REMARK 2. If T is finite, Proposition 1 ensures decidability of constraint satisfiability. In order to obtain a decision procedure, it is sufficient to instantiate the axioms of T and the axioms of equality (reflexivity, transitivity, symmetry, congruence) and to use a SAT-solver to decide constraint satisfiability. Alternatively, one can decide constraint satisfiability via congruence closure [13] and avoid instantiating the equality axioms. ◁

REMARK 3. Acyclity is a strong condition, often too strong. However, some condition must be imposed (otherwise we have undecidability, and then failure of finite model property, by reduction to word problem for finite presentations of monoids). In fact, the empty theory and the theory axiomatized by axiom 1 both have the finite model property even without acyciclity assumptions. ◁

We recall some basic definitions and notions from logic and model theory. We focus on the definitions of diagram, embedding, substructure and amalgamation.

We adopt the usual first-order syntactic notions of signature, term, atom, (ground) formula, sentence, and so on.

Let Σ be a first-order signature. The signature obtained from Σ by adding to it a set $\underline{a}$ of new constants (i.e., 0-ary function symbols) is denoted by $\Sigma^{\underline{a}}$. We indicate by $|\mathcal{A}|$ the support of a Σ-structure $\mathcal{A}$: this is the disjoint union of the sets $S^{\mathcal{A}}$, varying S among the sort symbols of $\mathcal{A}$. Analogously, given a Σ-structure $\mathcal{A}$, the signature Σ can be expanded to a new signature $\Sigma^{|\mathcal{A}|} := \Sigma \cup \{\bar{a} \mid a \in |\mathcal{A}|\}$ by adding a set of new constants $\bar{a}$ (the *name* for a), one for each element a in $\mathcal{A}$, with the convention that two distinct elements are denoted by different "name" constants. $\mathcal{A}$ can be expanded to a $\Sigma^{|\mathcal{A}|}$-structure $\mathcal{A}' := (\mathcal{A}, a)_{a \in |\mathcal{A}|}$ just interpreting the additional costants over the corresponding elements. From now on, when the meaning is clear from the context, we will freely use the notation $\mathcal{A}$ and $\mathcal{A}'$ interchangeably: in particular, given a Σ-structure $\mathcal{M}$ and a Σ-formula $\phi(\underline{x})$ with free variables that are all in $\underline{x}$, we will write, by abuse of notation, $\mathcal{A} \models \phi(\underline{a})$ instead of $\mathcal{A}' \models \phi(\underline{\bar{a}})$.

A Σ-*homomorphism* (or, simply, a homomorphism) between two Σ-structures $\mathcal{M}$ and $\mathcal{N}$ is any mapping $\mu : |\mathcal{M}| \longrightarrow |\mathcal{N}|$ among the support sets $|\mathcal{M}|$ of $\mathcal{M}$ and $|\mathcal{N}|$ of $\mathcal{N}$ satisfying the condition

$$\mathcal{M} \models \varphi \quad \Rightarrow \quad \mathcal{N} \models \varphi \tag{10}$$

for all $\Sigma^{|\mathcal{M}|}$-atoms φ (here $\mathcal{M}$ is regarded as a $\Sigma^{|\mathcal{M}|}$-structure, by interpreting each additional constant $a \in |\mathcal{M}|$ into itself and $\mathcal{N}$ is regarded as a $\Sigma^{|\mathcal{M}|}$-structure by interpreting each additional constant $a \in |\mathcal{M}|$ into $\mu(a)$). In case condition (10) holds for all $\Sigma^{|\mathcal{M}|}$-literals, the homomorphism μ is said to be an *embedding* and if it holds for all first order formulae, the embedding μ is said to be *elementary*. Notice the following facts:

(a) since we have equality in the signature, an embedding is an injective function;

(b) an embedding $\mu : \mathcal{M} \longrightarrow \mathcal{N}$ must be an algebraic homomorphism, that is for every n-ary function symbol f and for every $m_1, ..., m_n$ in $|\mathcal{M}|$, we must have $f^{\mathcal{N}}(\mu(m_1), ..., \mu(m_n)) = \mu(f^{\mathcal{M}}(m_1, ..., m_n))$;

(c) for an n-ary predicate symbol P we must have $(m_1, ..., m_n) \in P^{\mathcal{M}}$ iff $(\mu(m_1), ..., \mu(m_n)) \in P^{\mathcal{N}}$.

It is easily seen that an embedding $\mu : \mathcal{M} \longrightarrow \mathcal{N}$ can be equivalently defined as a map $\mu : |\mathcal{M}| \longrightarrow |\mathcal{N}|$ satisfying the conditions (a)-(b)-(c) above. If $\mu : \mathcal{M} \longrightarrow \mathcal{N}$ is an embedding which is just the identity inclusion $|\mathcal{M}| \subseteq |\mathcal{N}|$, we say that $\mathcal{M}$ is a *substructure* of $\mathcal{N}$ or that $\mathcal{N}$ is an *extension* of $\mathcal{M}$. A Σ-structure $\mathcal{M}$ is said to be *generated by* a set X included in its support $|\mathcal{M}|$ iff there are no proper substructures of $\mathcal{M}$ including X.

The notion of substructure can be equivalently defined as follows: given a Σ-structure $\mathcal{N}$ and a Σ-structure $\mathcal{M}$ such that $|\mathcal{M}| \subseteq |\mathcal{N}|$, we say that $\mathcal{M}$ is a Σ-*substructure* of $\mathcal{N}$ if:

- for every function symbol f inf Σ, the interpretation of f in $\mathcal{M}$ (denoted using $f^{\mathcal{M}}$) is the restriction of the interpretation of f in $\mathcal{N}$ to $|\mathcal{M}|$ (i.e. $f^{\mathcal{M}}(m) = f^{\mathcal{N}}(m)$ for every m in $|\mathcal{M}|$); this fact implies that a substructure $\mathcal{M}$ must be a subset of $\mathcal{N}$ which is closed under the application of $f^{\mathcal{N}}$.

- for every relation symbol P in Σ and every tuple $(m_1, ..., m_n) \in |\mathcal{M}|^n$, $(m_1, ..., m_n) \in P^{\mathcal{M}}$ iff $(m_1, ..., m_n) \in P^{\mathcal{N}}$, which means that the relation $P^{\mathcal{M}}$ is the restriction of $P^{\mathcal{N}}$ to the support of $\mathcal{M}$.

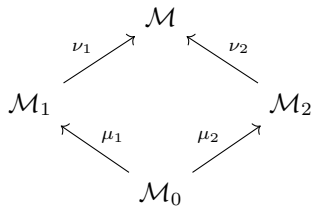
We recall that a substructure *preserves* and *reflects* validity of ground formulae, in the following sense: given a Σ-substructure $\mathcal{A}_1$ of a Σ-structure $\mathcal{A}_2$, a ground $\Sigma^{|\mathcal{A}_1|}$-sentence θ is true in $\mathcal{A}_1$ iff θ is true in $\mathcal{A}_2$.

Let $\mathcal{A}$ be a Σ-structure. The *diagram* of $\mathcal{A}$, denoted by $\Delta_\Sigma(\mathcal{A})$, is defined as the set of ground $\Sigma^{|\mathcal{A}|}$-literals (i.e. atomic formulae and negations of atomic formulae) that are true in $\mathcal{A}$. For the sake of simplicity, once again by abuse of notation, we will freely say that $\Delta_\Sigma(\mathcal{A})$ is the set of $\Sigma^{|\mathcal{A}|}$-literals which are true in $\mathcal{A}$.

An easy but nevertheless important basic result, called *Robinson Diagram Lemma* [21], says that, given any Σ-structure $\mathcal{B}$, the embeddings $\mu : \mathcal{A} \longrightarrow \mathcal{B}$ are in bijective correspondence with expansions of $\mathcal{B}$ to $\Sigma^{|\mathcal{A}|}$-structures which are models of $\Delta_\Sigma(\mathcal{A})$. The expansions and the embeddings are related in the obvious way: $\bar{a}$ is interpreted as $\mu(a)$.

Amalgamation is a classical algebraic concept. We give the formal definition of this notion.

DEFINITION 5 (AMALGAMATION). A theory T has the *amalgamation property* if for every couple of embeddings $\mu_1 : \mathcal{M}_0 \longrightarrow \mathcal{M}_1$, $\mu_2 : \mathcal{M}_0 \longrightarrow \mathcal{M}_2$ among models of T, there exists a model $\mathcal{M}$ of T endowed with embeddings $\nu_1 : \mathcal{M}_1 \longrightarrow \mathcal{M}$ and $\nu_2 : \mathcal{M}_2 \longrightarrow \mathcal{M}$ such that $\nu_1 \circ \mu_1 = \nu_2 \circ \mu_2$



◁

The triple $(\mathcal{M}, \mu_1, \mu_2)$ (or, by abuse, $\mathcal{M}$ itself) is said to be a T-amalgama of $\mathcal{M}_1, \mathcal{M}_2$ over $\mathcal{M}_0$

The following Lemma gives a useful folklore technique for finding model completions:

LEMMA 1. *Suppose that for every primitive Σ-formula $\exists x\, \phi(x, \underline{y})$ it is possible to find a quantifier-free formula $\psi(\underline{y})$ such that*

(i) $T \models \forall x \forall \underline{y}\, (\phi(x, \underline{y}) \rightarrow \psi(\underline{y}))$;

(ii) *for every model $\mathcal{M}$ of T, for every tuple of elements $\underline{a}$ from the support of $\mathcal{M}$ such that $\mathcal{M} \models \psi(\underline{a})$ it is possible to find another model $\mathcal{N}$ of T such that $\mathcal{M}$ embeds into $\mathcal{N}$ and $\mathcal{N} \models \exists x \phi(x, \underline{a})$.*

Then T has a model completion T^* axiomatized by the infinitely many sentences [19]

$$\forall \underline{y}\, (\psi(\underline{y}) \rightarrow \exists x\, \phi(x, \underline{y}))\ . \tag{11}$$

◁

PROOF. From (i) and (11) we clearly get that $T^\star$ admits quantifier elimination: in fact, in order to prove that a theory enjoys quantifier elimination, it is sufficient to teliminate quantifiers from *primitive* formulae (then the quantifier elimination for all formulae can be easily shown by an induction over their complexity). This is exactly what is guaranteed by (i) and (11).

Let $\mathcal{M}$ be a model of T. We show (by using a chain argument) that there exists a model $\mathcal{M}'$ of $T^\star$ such that $\mathcal{M}$ embeds into $\mathcal{M}'$. For every primitive formula $\exists x \phi(x, \underline{y})$, consider the set $\{(\underline{a}, \exists x \phi(x, \underline{a}))\}$ such that $\mathcal{M} \models \psi(\underline{a})$ (where ψ is related to ϕ as in (i)). By Zermelo's Theorem, the set $\{(\underline{a}, \exists x \phi(x, \underline{a}))\}$ can be well-ordered: let $\{(\underline{a}_i, \exists x \phi_i(x, \underline{a}_i))\}_{i \in I}$ be such a well-ordered set (where I is an ordinal). By transfinite induction on this well-order, we define $\mathcal{M}_0 := \mathcal{M}$ and, for each $i \in I$, $\mathcal{M}_{i+1}$ as the extension of $\mathcal{M}_i$ such that $\mathcal{M}_{i+1} \models \exists x \phi(x, \underline{y})$, which exists for (ii) since $\mathcal{M}_i \models \psi(a)$ (remember that validity of ground formulae is preserved passing through substructures and superstructures, and $\mathcal{M}_0 \models \psi(a)$).

Now we take the chain union $\mathcal{M}^1 := \bigcup_{i \in I} \mathcal{M}_i$: since T is universal, $\mathcal{M}^1$ is again a model of T, and it is possible to construct an analogous chain $\mathcal{M}^2$ as done above, starting from $\mathcal{M}^1$ instead of $\mathcal{M}$. Clearly, we get $\mathcal{M}_0 := \mathcal{M} \subseteq \mathcal{M}^1 \subseteq \mathcal{M}^2$ by construction. At this point, we iterate the same argument countably many times, so as to define a new chain of models of T:

$$\mathcal{M}_0 := \mathcal{M} \subseteq \mathcal{M}^1 \subseteq ... \subseteq \mathcal{M}^n \subseteq ...$$

Defining $\mathcal{M}' := \bigcup_n \mathcal{M}^n$, we trivially get that $\mathcal{M}'$ is a model of T such that $\mathcal{M} \subseteq \mathcal{M}'$ and satisfies all the sentences of type (11). The last fact can be shown using the following finiteness argument.

Fix ϕ, ψ as in (11). For every tuple $\underline{a}' \in \mathcal{M}'$ such that $\mathcal{M}' \models \psi(\underline{a}')$, by definition of $\mathcal{M}'$ there exists a natural number k such that $\underline{a}' \in \mathcal{M}^k$: since $\psi(a')$ is a ground formula, we get that also $\mathcal{M}^k \models \psi(\underline{a}')$. Therefore, we consider

[19] Notice that our T is assumed to be universal according to Definition 1, whereas T^* turns out to be universal-existential.

the step k of the countable chain: there, we have that the pair $(\underline{a}', \psi(\underline{a}'))$ appears in the enumeration given by the well-ordered set $\{(\underline{a}_i, \exists x \phi_i(x, \underline{a}_i))\}_{i \in I}$ (for such ordinal I) such that $\mathcal{M}^k \models \psi_i(\underline{a})$. Hence, by construction and since $\psi(\underline{a}')$ is a ground formula, we have that there exists a $j \in I$ such that $\mathcal{M}^k_j \models \psi(\underline{a}')$ and $\mathcal{M}^k_{j+1} \models \exists x \phi(x, \underline{a}')$. In conclusion, since the existential formulae are preserved passing to extensions, we obtain $\mathcal{M}' \models \exists x \phi(x, \underline{a}')$, as wanted. □

Proposition 2. *T has a model completion in case it is axiomatized by universal one-variable formulae and Σ is acyclic.*

Proof. We freely take inspiration from an analogous result in [45]. We preliminarly show that T is amalgamable. Then, for a suitable choice of ψ suggested by the acyclicity assumption, the amalgamation property will be used to prove the validy of the condition (ii) of Lemma 1: this fact (together with condition (i)) yields that T has a model completion which is axiomatized by the infinitely many sentences (11).

Let $\mathcal{M}_1$ and $\mathcal{M}_2$ two models of T with a submodel $\mathcal{M}_0$ of T in common (we suppose for simplicity that $|\mathcal{M}_1| \cap |\mathcal{M}_2| = |\mathcal{M}_0|$). We define a T-amalgam $\mathcal{M}$ of $\mathcal{M}_1, \mathcal{M}_2$ over $\mathcal{M}_0$ as follows (we use in an essential way the fact that Σ contains only *unary* function symbols). Let the support of $\mathcal{M}$ be the set-theoretic union of the supports of $\mathcal{M}_1$ and $\mathcal{M}_2$, i.e. $|\mathcal{M}| := |\mathcal{M}_1| \cup |\mathcal{M}_2|$. $\mathcal{M}$ has a natural Σ-structure inherited by the Σ-structures $\mathcal{M}_1$ and $\mathcal{M}_2$: for every function symbol f in Σ, we define, for each $m_i \in |\mathcal{M}_i| (i = 1, 2)$, $f^{\mathcal{M}}(m_i) := f^{\mathcal{M}_1}(m_i)$, i.e. the interpretation of f in $\mathcal{M}$ is the restriction of the interpretation of f in $\mathcal{M}_i$ for every element $m_i \in |\mathcal{M}_i|$. This is well-defined since, for every $a \in |\mathcal{M}_1| \cap |\mathcal{M}_2| = |\mathcal{M}_0|$, we have that $f^{\mathcal{M}}(a) := f^{\mathcal{M}_1}(a) = f^{\mathcal{M}_0}(a) = f^{\mathcal{M}_2}(a)$. It is clear that $\mathcal{M}_1$ and $\mathcal{M}_2$ are substructures of $\mathcal{M}$, and their inclusions agree on $\mathcal{M}_0$.

We show that the Σ-structure $\mathcal{M}$, as defined above, is a model of T. By hypothesis, T is axiomatized by universal one-variable formulae: so, we can consider T as a theory formed by axioms ϕ which are universal closures of clauses with just one variable, i.e. $\phi := \forall x (A_1(x) \wedge ... \wedge A_n(x) \rightarrow B_1(x) \vee ... \vee B_m(x))$, where A_j and B_k $(j = 1, ..., n$ and $k = 1, ..., m)$ are atoms.

We show that $\mathcal{M}$ satisfies all such formulae ϕ. In order to do that, suppose that, for every $a \in |\mathcal{M}|$, $\mathcal{M} \models A_j(a)$ for all $j = 1, ..., n$. If $a \in |\mathcal{M}_i|$, then $\mathcal{M} \models A_j(a)$ implies $\mathcal{M}_i \models A_j(a)$, since $A_j(a)$ is a ground formula. Since $\mathcal{M}_i$ is model of T and so $\mathcal{M}_i \models \phi$, we get that $\mathcal{M}_i \models B_k(a)$ for some $k = 1, ..., m$, which means that $\mathcal{M} \models B_k(a)$, since $B_k(a)$ is a ground formula. Thus, $\mathcal{M} \models \phi$ for every axiom ϕ of T, i.e. $\mathcal{M} \models T$ and, hence, $\mathcal{M}$ is a T-amalgam of $\mathcal{M}_1, \mathcal{M}_2$ over $\mathcal{M}_0$, as wanted

Now, given a primitive formula $\exists x \phi(x, \underline{y})$, we find a suitable ψ such that the hypothesis of Lemma 1 holds. We define $\psi(\underline{y})$ as the conjunction of the set of all quantifier-free $\chi(\underline{y})$-formulae such that $\phi(x, \underline{y}) \rightarrow \chi(\underline{y})$ is a logical consequences of T (they are finitely many - up to T-equivalence - because Σ is acyclic). By definition, clearly we have that (i) of Lemma 1 holds.

We show that also condition (ii) is satisfied. Let $\mathcal{M}$ be a model of T such that $\mathcal{M} \models \psi(\underline{a})$ for some tuple of elements $\underline{a}$ from the support of $\mathcal{M}$. Then, consider the Σ-substructure $\mathcal{M}[\underline{a}]$ of $\mathcal{M}$ generated by the elements $\underline{a}$: this substructure is finite (since Σ is acyclic), it is a model of T and we trivially have that $\mathcal{M}[\underline{a}] \models \psi(\underline{a})$, since $\psi(\underline{a})$ is a ground formula. In order to prove that there exists an extension $\mathcal{N}'$ of $\mathcal{M}[\underline{a}]$ such that $\mathcal{N} \models \exists x \phi(x, \underline{a})$, it is sufficient to prove (by the Robinson Diagram Lemma) that the $\Sigma^{|\mathcal{M}[\underline{a}]| \cup \{e\}}$-theory $\Delta(\mathcal{M}[\underline{a}]) \cup \{\phi(e, \underline{a})\}$ is T-consistent. For reduction to absurdity, suppose that the last theory is T-inconsistent. Then, there are finitely many literals $l_1(\underline{a}), ..., l_m(\underline{a})$ from $\Delta(\mathcal{M}[\underline{a}])$ (remember that $\Delta(\mathcal{M}[\underline{a}])$ is a finite set of literals since $\mathcal{M}[\underline{a}]$ is a finite structure) such that $\phi(e, \underline{a}) \models_T \neg(l_1(\underline{a}) \wedge ... \wedge l_m(\underline{a}))$. Therefore, defining $A(\underline{a}) := l_1(\underline{a}) \wedge ... \wedge l_m(\underline{a})$, we get that $\phi(e, \underline{a}) \models_T \neg A(\underline{a})$, which implies that $\neg A(\underline{a})$ is one of the $\chi(\underline{y})$-formulae appearing in $\psi(\underline{a})$. Since $\mathcal{M}[\underline{a}] \models \psi(\underline{a})$, we also have that $\mathcal{M}[\underline{a}] \models \neg A(\underline{a})$, which is a contraddiction: in fact, by definition of diagram, $\mathcal{M}[\underline{a}] \models A(\underline{a})$ must hold. Hence, there exists an extension $\mathcal{N}'$ of $\mathcal{M}[\underline{a}]$ such that $\mathcal{N}' \models \exists x \phi(x, \underline{a})$. Now, by amalgamation property, there exists a T-amalgam $\mathcal{N}$ of $\mathcal{M}$ and $\mathcal{N}'$ over $\mathcal{M}[\underline{a}]$: clearly, $\mathcal{N}$ is an extension of $\mathcal{M}$ and, since $\mathcal{N}' \hookrightarrow \mathcal{N}$ and $\mathcal{N}' \models \exists x \phi(x, \underline{a})$, also $\mathcal{N} \models \exists x \phi(x, \underline{a})$ holds, as required.

Remark 4. The proof of Proposition 2 gives an algorithm for quantifier elimination in the model completion. The algorithm works as follows (see the formula (11)): to eliminate the quantifier x from $\exists x\, \phi(x, \underline{y})$ take the conjunction of the clauses $\chi(\underline{y})$ implied by $\phi(x, \underline{y})$. This algorithm is not practical: better algorithms can be obtained by using Knuth-Bendix procedure, as we shall show in the forthcoming paper [18]. ◁

C. PROOFS AND COMPLEMENTS FOR SECTION 5

In this section we present Theorems 5 and 6 that constitute the proof of Theorem 1 from Section 5.

Theorem 5. *Let $\langle \Sigma, T \rangle$ be a DB schema. Then, for any a simple artifact system like in Definition 3 $\mathcal{S}$ with $\langle \Sigma, T \rangle$ as its DB schema, backward search algorithm is effective and partially correct for solving safety problems for $\mathcal{S}$. If, in addition, Σ is acyclic, backward search terminates and decides safety problems for $\mathcal{S}$.* ◁

PROOF. recall formula (3)

$$\iota(\underline{x}^0) \wedge \tau(\underline{x}^0, \underline{x}^1) \wedge \cdots \wedge \tau(\underline{x}^{k-1}, \underline{x}^k) \wedge \upsilon(\underline{x}^k) \quad .$$

By definition, $\mathcal{S}$ is unsafe iff for some n, the formula (3) is satisfiable in a DB-instance of $\langle \Sigma, T \rangle$. Thanks to Assumption 1, T has the finite model property and consequently, as (3) is an existential Σ-formula, $\mathcal{S}$ is unsafe iff for some n, formula (3) is satisfiable in a model of T; furthermore, again by Assumption 1, $\mathcal{S}$ is unsafe iff for some n, formula (3) is satisfiable in a model of T^*. Thus, we shall concentrate on satisfiability in models of T^* in order to prove the Theorem.

Let us call B_n (resp. ϕ_n) the status of the variable B (resp. ϕ) after n executions in line 4 (resp. line 6) of Algorithm 1. Notice that we have $T^* \models \phi_{j+1} \leftrightarrow Pre(\tau, \phi_j)$ for all j and that

$$T \models B_n \leftrightarrow \bigvee_{0 \leq j < n} \phi_j \tag{12}$$

is an invariant of the algorithm.

Since we are considering satisfiability in models of T^*, we can apply quantifier elimination and so the satisfiability of (3) is equivalent to the satisfiability of $\iota \wedge \phi_n$: this is a quantifier-free formula (because in line 6 of Algorithm 1), whose satisfiability (wrt T or equivalently wrt T^*)[20] is decidable by Assumption 1, so if Algorithm 1 terminates with an unsafe outcome, then $\mathcal{S}$ is really unsafe.

Now consider the satisfiability test in line 2. This is again a satisfiability test for a quantifier-free formula, thus it is decidable. In case of a safe outcome, we have that $T \models \phi_n \rightarrow B_n$; this means that, if we could continue executing the loop of Algorithm 1, we would nevertheless get $T^* \models B_m \leftrightarrow B_n$ for all $m \geq n$.[21] This would entail that $\iota \wedge \phi_m$ is always unsatisfiable (because of (12) and because $\iota \wedge \phi_j$ was unsatisfiable for all $j < n$), which is the same (as remarked above) as saying that all formulae (3) are unsatisfiable. Thus $\mathcal{S}$ is safe.

In case Σ is acyclic, there are only finitely many quantifier-free formulae (in which the finite set of variables $\underline{x}$ occur), so it is evident that the algorithm must terminate: because of (12), the unsatisfiability test of Line 2 must eventually succeed, if the unsatisfiability test of Line 3 never does so. □

For complexity questions, we have the following result:

THEOREM 6. *Let Σ be an acyclic DB signature and $\langle \Sigma, T \rangle$ a DB schema built on top of it. Then, for every SAS $\mathcal{S} = \langle \Sigma, T, \underline{x}, \iota, \tau \rangle$, deciding safety problems for $\mathcal{S}$ is in PSPACE in the size of $\underline{x}$, of ι and of τ.* ◁

PROOF. We need to modify Algorithm 1 (we make it nondeterministic and use Savitch's Theorem saying that PSPACE = NPSPACE).

Since Σ is acyclic, there are only finitely many terms involving a single variable, let this number be k_Σ (we consider T, Σ and hence k_Σ constant for our problems). Then, since all function symbols are unary, it is clear that we have at most $2^{O(n^2)}$ conjunctions of sets of literals involving at most n variables and that if the system is unsafe, unsafety can be detected with a run whose length is at most $2^{O(n^2)}$. Thus we introduce a counter to be incremented during the main loop (lines 2-6) of Algorithm 1. The fixpoint test in line 2 is removed and loop is executed only until the maximum length of an unsafe run is not exceeded (notice that an exponential counter requires polynomial space).

Inside the loop, line 4 is removed (we do not need anymore the variable B) and line 6 is modified as follows. We replace line 6 of the algorithm by

$$6'. \quad \phi \longleftarrow \alpha(\underline{x});$$

where α is a non-deterministically chosen conjunction of literals implying $\mathsf{QE}(T^*, \phi)$. Notice that to check the latter, there is no need to compute $\mathsf{QE}(T^*, \phi)$: recalling the proof of Proposition 2 and Remark 4 it is sufficient to check that $T \models \alpha \rightarrow C$ holds for every clause $C(\underline{x})$ such that $T \models \phi \rightarrow C$.

The algorithm is now in PSPACE, because all the satisfiability tests we need are, as a consequence of the proof of Proposition 1, in NP: all such tests are reducible to T-satisfiability tests for quantifier-free Σ-formulae involving the variables $\underline{x}$ and the additional (skolemized) quantified variables occurring in the transitions [22]. In fact, all these satisfiability tests are applied to formulae whose length is polynomial in the size of $\underline{x}$, of ι and of τ. □

[20] T-satisfiability and T^*-satisfiability are equivalent, by the definition of T^*, as far as existential (in particular, quantifier-free) formulae are concerned.

[21] In more detail: recall the invariant (12) and that $T^* \models \phi_{j+1} \leftrightarrow Pre(\tau, \phi_j)$ holds for all j. Thus, from $T \models \phi_n \rightarrow B_n$, we get $T \models \phi_{n+1} \rightarrow Pre(\tau, B_n)$; since Pre commutes with disjunctions, we have $T^* \models Pre(\tau, B_n) \leftrightarrow \bigvee_{1 \leq j \leq n} \phi_j$. Now (using $T \models \phi_n \rightarrow B_n$ again), we get $T^* \models \phi_{n+1} \rightarrow B_n$, that is $T^* \models B_{n+1} \leftrightarrow B_n$. Since then $T^* \models \phi_{n+1} \rightarrow B_{n+1}$, we can repeat the argument for all $m \geq n$.

[22] For the test in line 3, we just need replace in ϕ the $\underline{x}$ by their values given by ι, conjoin the result with all the ground instances of the axioms of T and finally decide satisfiability with congruence closure algorithm of a polynomial size ground conjunction of literals.

D. PROOFS FROM SECTION 6

The technique used for proving Theorem 2 is similar to that used in [20] (but here we have to face some additional complications, due to the fact that our quantifier elimination is not directly available, it is only indirectly available via model completions).

When introducing our transition formulae in (2), (6) we made use of definable extensions and also of some function definitions via λ-abstraction. We already observed that such uses are due to notational convenience and do not really go beyond first-order logic. We are clarifying one more point now, before going into formal proofs. The lambda-abstraction definitions in (6) will make the proof of Lemma 2 below smooth. Recall that an expression like

$$b = \lambda y.F(y, \underline{z})$$

can be seen as a mere abbreviation of $\forall y\ b(y) = F(y, \underline{z})$. However, the use of such abbreviation makes clear that e.g. a formula like

$$\exists b\ (b = \lambda y.F(y, \underline{z}) \wedge \phi(\underline{z}, b))$$

is equivalent to

$$\phi(\underline{z}, \lambda y.F(y, \underline{z})/b) \quad . \tag{13}$$

Since our $\phi(\underline{z}, b)$ is in fact a first-order formula, our b can occur in it only in terms like $b(t)$, so that in (13) all occurrences of λ can be eliminated by the so-called β-conversion: replace $\lambda y F(y, \underline{z})(t)$ by $F(t, \underline{z})$. Thus, in the end, either we use definable extensions or definitions via lambda abstractions, *the formulae we manipulate can always be converted into plain first-order* Σ- *or* Σ_{ext}*-formulae.*

Let us call *extended state formulae* the formulae of the kind $\exists \underline{e}\ \phi(\underline{e}, \underline{x}, \underline{a})$, where ϕ is quantifier-free and the $\underline{e}$ are individual variables of both artifact and basic sorts.

LEMMA 2. *The preimage of an extended state formula is logically equivalent to an extended state formula.* ◁

PROOF. We manipulate the formula

$$\exists \underline{x}'\ \exists \underline{a}'\ (\tau(\underline{x}, \underline{a}, \underline{x}', \underline{a}') \wedge \exists \underline{e}\ \phi(\underline{e}, \underline{x}', \underline{a}')) \tag{14}$$

up to logical equivalence, where τ is given by[23]

$$\exists \underline{e}_0\ (\gamma(\underline{e}_0, \underline{x}, \underline{a}) \wedge \underline{x}' = \underline{F}(\underline{e}_0, \underline{x}, \underline{a}) \wedge \underline{a}' = \lambda y.\underline{G}(y, \underline{e}_0, \underline{x}, \underline{a})) \tag{15}$$

(here we used plain equality for conjunctions of equalities, e.g. $\underline{x}' = \underline{F}(\underline{e}_0, \underline{x}, \underline{a})$ stands for $\bigwedge_i x'_i = F_i(\underline{e}, \underline{x}, \underline{a})$). Repeated substitutions show that (14) is equivalent to

$$\exists \underline{e}\ \exists \underline{e}_0\ (\gamma(\underline{e}_0, \underline{x}, \underline{a}) \wedge \phi(\underline{e}, \underline{F}(\underline{e}_0, \underline{x}, \underline{a})/\underline{x}', \lambda y.\underline{G}(y, \underline{e}_0, \underline{x}, \underline{a})/\underline{a}')) \tag{16}$$

which is an extended state formula. □

LEMMA 3. *For every extended state formula there is a state formula equivalent to it in all* Σ_{ext}*-models of* T^*. ◁

PROOF. Let $\exists \underline{e}\, \exists \underline{y}\ \phi(\underline{e}, \underline{y}, \underline{x}, \underline{a})$, be an extended state formula, where ϕ is quantifier-free, the $\underline{e}$ are variables whose sort is an artifact sort and the $\underline{y}$ are variables whose sort is a basic sort.

Now observe that, according to our definitions, the artifact components have an artifact sort as source sort and a basic sort as target sort; since equality is the only predicate, the literals in ϕ can be divided into equalities/inequalities between variables from $\underline{e}$ and literals where the $\underline{e}$ can only occur as arguments of an artifact component. Let $\underline{a}[\underline{e}]$ be the tuple of the terms among the terms of the kind $a_j[e_s]$ which are well-typed; using disjunctive normal forms, our extended state formula can be written as a disjunction of formulae of the kind

$$\exists \underline{e}\, \exists \underline{y}\ (\phi_1(\underline{e}) \wedge \phi_2(\underline{y}, \underline{x}, \underline{a}[\underline{e}]/\underline{z})) \tag{17}$$

where ϕ_1 is a conjunction of equalities/inequalities, $\phi_2(\underline{y}, \underline{x}, \underline{z})$ is a quantifier-free Σ-formula and $\phi_2(\underline{y}, \underline{x}, \underline{a}[\underline{e}]/\underline{z})$ is obtained from ϕ_2 by replacing the variables $\underline{z}$ by the terms $\underline{a}[\underline{e}]$. Moving inside the existential quantifiers $\underline{y}$, we can rewrite (17) to

$$\exists \underline{e}\ (\phi_1(\underline{e}) \wedge \exists \underline{y}\, \phi_2(\underline{y}, \underline{x}, \underline{a}[\underline{e}]/\underline{z})) \tag{18}$$

Since T^* has quantifier elimination, we have that there is $\psi(\underline{x}, \underline{z})$ which is equivalent to $\exists \underline{y}\, \phi_2(\underline{y}, \underline{x}, \underline{z}))$ in all models of T^*; thus in all Σ_{ext}-models of T^*, the formula (18) is equivalent to

$$\exists \underline{e}\ (\phi_1(\underline{e}) \wedge \psi(\underline{x}, \underline{a}[\underline{e}]/\underline{z}))$$

which is a state formula. □

We underline that Lemmas 2 and 3 both give an explicit effective procedure for computing equivalent (extended) state formulae. Used one after the other, such procedures extends the procedure $QE(T^*, \phi)$ in line 6 of Algorithm 1

[23] Actually, τ is a disjunction of such formulae, but it easily seen that disjunction can be accommodated by moving existential quantifiers back-and-forth through them.

to (non simple) artifact systems. Thanks to such procedure, the only formulae we need to test for satisfiability in lines 2 and 3 of the backward reachability algorithm are the $\exists\forall$-formulae introduced below.

Let us call $\exists\forall$-formulae the formulae of the kind

$$\exists \underline{e}\ \forall \underline{i}\ \phi(\underline{e}, \underline{i}, \underline{x}, \underline{a}) \tag{19}$$

where the variables $\underline{e}, \underline{i}$ are variables whose sort is an artifact sort and ϕ is quantifier-free. The crucial point for the following lemma to hold is that the *universally* quantified variables in $\exists\forall$-formulae are all of artifact sorts:

LEMMA 4. *The satisfiability of a $\exists\forall$-formula in a Σ_{ext}-model of T is decidable; moreover, a $\exists\forall$-formula is satisfiable in a Σ_{ext}-model of T iff it is satisfiable in a DB-instance of $\langle \Sigma_{ext}, T\rangle$ iff it is satisfiable in a Σ_{ext}-model of T^*.* $\triangleleft$

PROOF. First of all, notice that a $\exists\forall$-formula (19) is equivalent to a disjunction of formulae of the kind

$$\exists \underline{e}\ (\mathrm{Diff}(\underline{e}) \wedge \forall \underline{i}\ \phi(\underline{e}, \underline{i}, \underline{x}, \underline{a})) \tag{20}$$

where $\mathrm{Diff}(\underline{e})$ says that any two variables of the same sort from the $\underline{e}$ are distinct (to this aim, it is sufficient to guess a partition and to keep, via a substitution, only one element for each equivalence class).[24] So we can freely assume that $\exists\forall$-formulae are all of the kind (20).

Now, by the way Σ_{ext} is built, the only atoms occurring in ϕ whose arguments involve terms of artifact sorts are of the kind $e_s = e_j$, so all such atoms can be replaced either by $\top$ or by $\bot$ (depending on whether we have $s = j$ or not). So we can assume that there are no such atoms in ϕ and as a result, the variables $\underline{e}$, $\underline{i}$ can only occur as arguments of the $\underline{a}$.

Let us consider now the set of all (sort-matching) substitutions σ mapping the $\underline{i}$ to the $\underline{e}$. The formula (20) is satisfiable (respectively: in a Σ_{ext}-model of T, in a DB-instance of $\langle \Sigma_{ext}, T\rangle$, in a Σ_{ext}-model of T^*) iff so it is the formula

$$\exists \underline{e}\ (\mathrm{Diff}(\underline{e}) \wedge \bigwedge_{\sigma} \phi(\underline{e}, \underline{i}\sigma, \underline{x}, \underline{a})) \tag{21}$$

(here $\underline{i}\sigma$ means the componentwise application of σ to the $\underline{i}$): this is because, if (21) is satisfiable in $\mathcal{M}$, then we can take as $\mathcal{M}'$ the same Σ_{ext}-structure as $\mathcal{M}$, but with the interpretation of the artifact sorts restricted only to the elements named by the $\underline{e}$ and get in this way a Σ_{ext}-structure $\mathcal{M}'$ satisfying (20) (notice that $\mathcal{M}'$ is still a DB-instance of $\langle \Sigma_{ext}, T\rangle$ or a Σ_{ext}-model of T^*, if so was $\mathcal{M}$). Thus, we can freely concentrate on the satisfiability problem of formulae of the kind (21) only.

Let now $\underline{a}[\underline{e}]$ be the tuple of the terms among the terms of the kind $a_j[e_s]$ which are well-typed. Since in (21) the $\underline{e}$ can only occur as arguments of the artifact components, as observed above, the formula (21) is in fact of the kind

$$\exists \underline{e}\ (\mathrm{Diff}(\underline{e}) \wedge \psi(\underline{x}, \underline{a}[\underline{e}]/\underline{z})) \tag{22}$$

where $\psi(\underline{x}, \underline{z})$ is a quantifier-free Σ-formula and $\psi(\underline{x}, \underline{a}[\underline{e}]/\underline{z})$ is obtained from ψ by replacing the variables $\underline{z}$ by the terms $\underline{a}[\underline{e}]$ (notice that the $\underline{z}$ are of basic sorts because the target sorts of the artifact components are basic sorts).

It is now evident that (22) is satisfiable (respectively: in a Σ_{ext}-model of T, in a DB-instance of $\langle \Sigma_{ext}, T\rangle$, in a Σ_{ext}-model of T^*) iff the formula

$$\psi(\underline{x}, \underline{z}) \tag{23}$$

is satisfiable (respectively: in a Σ-model of T, in a DB-instance of $\langle \Sigma, T\rangle$, in a Σ-model of T^*). In fact, if we are given a Σ-structure $\mathcal{M}$ and an assignment satisfying (23), we can easily expand $\mathcal{M}$ to a Σ_{ext}-structure by taking the e's themselves as the elements of the interpretation of the artifact sorts; in the so-expanded Σ_{ext}-structure, we can interpret the artifact components $\underline{a}$ by taking the $\underline{a}[\underline{e}]$ to be the elements assigned to the $\underline{z}$ in the satisfying assignment for (23).

Thanks to Assumption 1, the satisfiability of (23) in a Σ-model of T, in a DB-instance of $\langle \Sigma, T\rangle$, or in a Σ-model of T^* are all equivalent and decidable. $\square$

The instantiation algorithm of Lemma 4 can be used to discharge the satisfiability tests in lines 2 and 3 of Algorithm 1 because the conjunction of a state formula and of the negation of a state formula is a $\exists\forall$-formula (notice that ι is itself the negation of a state formula, according to (4)).

Theorem 2 *The backward search algorithm (cf. Algorithm 1), applied to artifact systems, is effective and partially correct.*

PROOF. Recall that $\mathcal{S}$ is unsafe iff there is no DB-instance $\mathcal{M}$ of $\langle \Sigma_{ext}, T\rangle$, no $k \geq 0$ and no assignment in $\mathcal{M}$ to the variables $\underline{x}^0, \underline{a}^0 \ldots, \underline{x}^k, \underline{a}^k$ such that the formula (7)

$$\iota(\underline{x}^0, \underline{a}^0) \wedge \tau(\underline{x}^0, \underline{a}^0, \underline{x}^1, \underline{a}^1) \wedge \cdots \wedge \tau(\underline{x}^{k-1}, \underline{a}^{k-1}, \underline{x}^k, \underline{a}^k) \wedge \upsilon(\underline{x}^k, \underline{a}^k)$$

is true in $\mathcal{M}$. It is sufficient to show that this is equivalent to saying that there is no Σ_{ext}-model $\mathcal{M}$ of T^*, no $k \geq 0$

[24] In the MCMT implementation, state formulae are always maintained so that all existential variables occurring in them are differentiated, so that there is no need of this expensive computation step.

and no assignment in $\mathcal{M}$ to the variables $\underline{x}^0, \underline{a}^0 \ldots, \underline{x}^k, \underline{a}^k$ such that (7) is true in $\mathcal{M}$ (once this is shown, the proof goes in the same way as the proof of Theorem 1).

Now, the formula (7) is satisfiable in a Σ_{ext}-structure $\mathcal{M}$ under a suitable assignment iff the formula

$$\iota(\underline{x}^0, \underline{a}^0) \ \wedge \ \exists \underline{a}^1 \exists \underline{x}^1 (\tau(\underline{x}^0, \underline{a}^0, \underline{x}^1, \underline{a}^1) \wedge \cdots \\ \cdots \wedge \exists \underline{a}^k \exists \underline{x}^k (\tau(\underline{x}^{k-1}, \underline{a}^{k-1}, \underline{x}^k, \underline{a}^k) \wedge \upsilon(\underline{x}^k, \underline{a}^k)) \cdots)$$

is satisfiable in $\mathcal{M}$ under a suitable assignment; by Lemma 2, the latter is equivalent to a formula of the kind

$$\iota(\underline{x}, \underline{a}) \ \wedge \ \exists \underline{e} \exists \underline{z} \, \phi(\underline{e}, \underline{z}, \underline{x}, \underline{a}) \tag{24}$$

where $\exists \underline{e} \exists \underline{z} \, \phi(\underline{e}, \underline{z}, \underline{x}, \underline{a})$ is an extended state formula (thus ϕ is quantifier-free, the $\underline{e}$ are variables of artifact sorts and the $\underline{z}$ are variables of basic sorts - we renamed $\underline{x}^0, \underline{a}^0$ as $\underline{x}, \underline{a}$). However the satisfiability of (24) is the same as the satisfiability of $\exists \underline{e} \, (\iota(\underline{x}, \underline{a}) \wedge \phi(\underline{e}, \underline{z}, \underline{x}, \underline{a}))$; the latter, in view of (4), is a $\exists\forall$-formula and so Lemma 4 applies and shows that its satisfiability in a DB-instance of $\langle \Sigma_{ext}, T \rangle$ is the same as its satisfiability in a Σ_{ext}-model of T^*. □

E. PROOFS FROM SECTION 7

We begin by recalling some basic facts about wqo's. Recall that a *well-quasi-order* (wqo) is a set W endowed with a reflexive-transitive relation $\leq$ having the following property: for every infinite succession

$$w_0, w_1, \ldots, w_i, \ldots$$

of elements from W there are i, j such that $i < j$ and $w_i \leq w_j$.

The fundamental result about wqo's is the following result, which is a consequence of the well-known Kruskal's Tree Theorem [35]:

THEOREM 7. *If $(W, \leq)$ is a wqo, then so is the partial order of the finite lists over W, ordered by componentwise subword comparison (i.e. $w \leq w'$ iff there is a subword w_0 of w' of the same length as w, such that the i-th entry of w is less or equal to - in the sense of $(W, \leq)$ - the i-th entry of w_0, for all $i = 0, \ldots |w|$).* ◁

Various wqo's can be recognized by applying the above Theorem; in particular, the Theorem implies that the cartesian product of wqos is a wqo. As an application, notice that $\mathbb{N}$ is a wqo, hence the following Corollary (known as Dikson Lemma) follows:

COROLLARY 1. *The cartesian product of k-copies of $\mathbb{N}$ (and also of $\mathbb{N} \cup \{\infty\}$), with componentwise ordering, is a wqo.* ◁

Let us now turn to the terminology introduced in Subsection 7.1 and in particular to the numbers $k_1(\mathcal{M}), \ldots, k_N(\mathcal{M}) \in \mathbb{N} \cup \{\infty\}$ counting the numbers of elements generating (as singletons) the cyclic substructures $\mathcal{C}_1, \ldots, \mathcal{C}_N$, respectively (we assume the acyclicity of Σ and consequently also of $\tilde{\Sigma}$).

LEMMA 5. *Let $\mathcal{M}, \mathcal{N}$ be $\tilde{\Sigma}$-structures. If the inequalities*

$$k_1(\mathcal{M}) \leq k_1(\mathcal{N}), \ldots, k_N(\mathcal{M}) \leq k_N(\mathcal{N})$$

hold, then all local formulae true in $\mathcal{M}$ are also true in $\mathcal{N}$. ◁

PROOF. Notice that local formulae (viewed in $\tilde{\Sigma}$) are sentences, because they do not have free variable occurrences - the $\underline{a}, \underline{x}$ are now constant function symbols and individual constants, respectively. The proof of the Lemma is fairly obvious: notice that, once we assigned some $\alpha(e_i)$ in $\mathcal{M}$ to the variable e_i, the truth of a formula like $\phi(e_i, \underline{x}, \underline{a})$ under such an assignment depends only on the $\tilde{\Sigma}$-substructure generated by $\alpha(e_i)$, because ϕ is quantifier-free and e_i is the only $\tilde{\Sigma}$-variable occurring in it. In fact, if a local state formula $\exists e_1 \cdots \exists e_k \left(\delta(e_1, \ldots, e_k) \wedge \bigwedge_{i=1}^k \phi_i(e_i, \underline{x}, \underline{a}) \right)$ is true in $\mathcal{M}$, then there exist elements $\bar{e}_1, \cdots, \bar{e}_n$ (in the interpretation of some artifact sorts), each of which makes ϕ_i true. Hence, ϕ_i is also true in the corresponding cyclic structure generated by $\bar{e}_i$. Since $k_1(\mathcal{M}) \leq k_1(\mathcal{N}), \ldots, k_N(\mathcal{M}) \leq k_N(\mathcal{N})$ hold, then also in $\mathcal{N}$ there are at least as many elements in the interpretation of artifact sorts as there are in $\mathcal{M}$ that validate all the ϕ_i . Thus, we get that $\exists e_1 \cdots \exists e_k \left(\delta(e_1, \ldots, e_k) \wedge \bigwedge_{i=1}^k \phi_i(e_i, \underline{x}, \underline{a}) \right)$ is true also in $\mathcal{N}$, as wanted. □

Theorem 3 *If Σ is acyclic, the backward search algorithm (cf. Algorithm 1) terminates when applied to a local safety formula in an artifact transition system, whose transition formula is a disjunction of local transition formulae.*

PROOF. Suppose the algorithm does not terminate. Then the fixpoint test of Line 2 fails infinitely often. Recalling that the T-equivalence of B_n and of $\bigvee_{0 \leq j < n} \phi_j$ is an invariant of the algorithm (here ϕ_n, B_n are the status of the variables ϕ, B after n execution of the main loop), this means that there are models

$$\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_i, \ldots$$

such that for all i, we have that $\mathcal{M}_i \models \phi_i$ and $\mathcal{M}_i \not\models \phi_j$ (all $j < i$). But the ϕ_i are all local formulae, so considering the tuple of cardinals $k_1(\mathcal{M}_i), \ldots, k_N(\mathcal{M}_i)$ and Lemma 5, we get a contradiction, in view of Dikson Lemma. This is because, by Dikson Lemma, $(\mathbb{N} \cup \{\infty\})^N$ is a wqo, so there exist i, j such that $j < i$ and $k_1(\mathcal{M}_j) \leq k_1(\mathcal{M}_i), \ldots, k_N(\mathcal{M}_j) \leq k_N(\mathcal{M}_i)$. Using Lemma 5, we get that ϕ_j, which is local and true in $\mathcal{M}_j$, is also true in $\mathcal{M}_i$, which is a contradiction. □

The proof of Theorem 4 is more complex, but follows a similar schema[25]. If $(W, \leq)$ is a partial order, we consider the set $M(W)$ of finite multisets of W as a partial order in the following way:[26] say that $M \leq N$ holds iff there is an injection $p : M \longrightarrow N$ such that $m \leq p(m)$ holds for all $m \in M$ (of course, the notion of an injection should take care of multiplicities: p should associate to every occurrence of m an occurrence $p(m)$ of an element of N so that different elements/different occurrences are associated to different elements/different occurrences).

COROLLARY 2. *If $(W, \leq)$ is a wqo, then so is $(M(W), \leq)$ as defined above.* ◁

PROOF. This is due to the fact that one can convert a multiset M to a list $L(M)$ so that if $L(M) \leq L(N)$ holds, then also $M \leq N$ holds (such a conversion L can be obtained by ordering the occurrences of elements in M in any arbitrarily chosen way). □

We assume that the graph $G(\tilde{\Sigma})$ associated to $\tilde{\Sigma}$ is a tree (the generalization to the case where such a graph is a forest is trivial). This means in particular that each sort is the domain of at most one function symbol and that there just one sort which is not the domain of any function symbol (let us call it the *root sort* of $\tilde{\Sigma}$ and let us denote it with S_r).

By induction on the height[27] of a sort S in the above graph, we define a wqo $w(S)$ (in the definition we use the fact the cartesian product of wqo's is a wqo and Corollary 2). Let $S_1, \ldots, S_n$ be the sons of S in the tree; put

$$w(S) \ := \ M(w(S_1)) \times \cdots \times M(w(S_n)) \tag{25}$$

(thus, if S is a leaf, $w(S)$ is the trivial one-element wqo - its only element is the empty tuple).

Let now $\mathcal{M}$ be a finite $\tilde{\Sigma}$-structure; we indicate with $S^{\mathcal{M}}$ the interpretation in $\mathcal{M}$ of the sort S (it is a finite set). For $a \in S^{\mathcal{M}}$, we define the multiset $M_{\mathcal{M}}(a) \in w(S)$, again by induction on the height of S. Suppose that $S_1, \ldots, S_n$ are the sons of S and that the arc from S_i to S is labeled by the function symbol f_i; then we put

$$\begin{aligned} M_{\mathcal{M}}(a) \ := \ & \langle \{M_{\mathcal{M}}(b_1) \mid b_1 \in S_1^{\mathcal{M}} \text{ and } f_1^{\mathcal{M}}(b_1) = a\}, \ldots \\ & \ldots, \{M_{\mathcal{M}}(b_n) \mid b_n \in S_n^{\mathcal{M}} \text{ and } f_n^{\mathcal{M}}(b_n) = a\} \rangle \end{aligned}$$

where $f_i^{\mathcal{M}}$ $(i = 1, \ldots, n)$ is the interpretation of the symbol f_i in $\mathcal{M}$.

Moreover, for every sort S, we let

$$M_{\mathcal{M}}(S) \ := \ \{M_{\mathcal{M}}(a) \mid a \in S^{\mathcal{M}}\} \quad . \tag{26}$$

Finally, we define

$$M(\mathcal{M}) \ := \ M_{\mathcal{M}}(S_r) \quad . \tag{27}$$

For termination, the relevant Lemma is the following:

LEMMA 6. *Given two finite models $\mathcal{M}$ and $\mathcal{N}$, we have that if $M(\mathcal{M}) \leq M(\mathcal{N})$, then $\mathcal{M}$ embeds into $\mathcal{N}$.* ◁

PROOF. Again, we make an induction on the height of S, proving the claim for the subsignature of $\tilde{\Sigma}$ having S as a root (let us call this the S-subsignature).

Let $\mathcal{M}$ be a model over the S-subsignature. For every $a \in S^{\mathcal{M}}$, and for every $f_i : S_i \longrightarrow S$, if we restrict $\mathcal{M}$ to the elements in the f_i-fibers of a, we get a model $\mathcal{M}_{f_i,a}$ for the S_i-subsignature (an element $c \in \tilde{S}^{\mathcal{M}}$ is in the f_i-fiber of a if, taking the term t corresponding to the composition of the functions symbols going from $\tilde{S}$ to S_i, we have that $f_i^{\mathcal{M}}(t^{\mathcal{M}}(c)) = a$). In addition, if $M_{\mathcal{M}}(a) = (M_1, \ldots, M_n)$, then $M_i = M(\mathcal{M}_{f_i,a})$ by definition. Finally, observe that the restriction of $\mathcal{M}$ to the S_i-subsignature is the disjoint union of the f_i-fibers models $\mathcal{M}_{f_i,a}$, varying $a \in S^{\mathcal{M}}$.

Suppose now that $\mathcal{M}, \mathcal{N}$ are models over the S-subsignature such that $M(\mathcal{M}) \leq M(\mathcal{N})$; this means that we can find an injective map μ mapping $S^{\mathcal{M}}$ into $S^{\mathcal{N}}$ so that $M_{\mathcal{M}}(a) \leq M_{\mathcal{N}}(\mu(a))$. If $M_{\mathcal{M}}(a) = (M_1, \ldots, M_n)$ and $M_{\mathcal{N}}(\mu(a)) = (N_1, \ldots, N_n)$, we then have that $M_i \leq N_i$ for every $i = 1, \ldots, n$. Considering that, as noticed above, $M_i = \mathcal{M}_{f_i,a}$ and $N_i = \mathcal{N}_{f_i,\mu(a)}$, by induction hypothesis, we have embeddings $\nu_{i,a}$ for the f_i-fibers models of a and $\mu(a)$ (for every $a \in S^{\mathcal{M}}$ and $i = 1, \ldots, n$). Glueing these embeddings to the disjoint union (varying i, a) and adding them μ as S-component, we get the desired embedding of $\mathcal{M}$ into $\mathcal{N}$. □

[25] For simplicity, we give the argument for the case where we do not have constants and artifact variables (footnote 28 shows how to extend the argument to the general case).

[26] This is not the canonical ordering used for multisets, see eg [9].

[27] This is defined as the length of the longest path from S to a leaf.

PROPOSITION 3. *If $\tilde{\Sigma}$ is tree-like, then the finite $\tilde{\Sigma}$-structures are a wqo with respect to the embeddability quasi-order.*$\triangleleft$

PROOF. An immediate consequence of the previous lemma. $\square$

Theorem 4 *The backward search algorithm (cf. Algorithm 1) terminates when applied to an artifact transition system whose artifact setting is tree-like.*

PROOF. Similarly to the proof of Theorem 3, suppose the algorithm does not terminate. Then the fixpoint test of Line 2 fails infinitely often. Recalling that the T-equivalence of B_n and of $\bigvee_{0\leq j<n}\phi_j$ is an invariant of the algorithm (here ϕ_n, B_n are the status of the variables ϕ, B after n execution of the main loop), this means that there are models

$$\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_i, \ldots$$

such that for all i, we have that $\mathcal{M}_i \models \phi_i$ and $\mathcal{M}_i \not\models \phi_j$ (all $j < i$). The models can be taken to be all finite, by Lemma 4. But the ϕ_i are all existential sentences in $\tilde{\Sigma}$, so this is incompatible to the fact that, by Proposition 3, there are $j < i$ with $\mathcal{M}_j$ embeddable into $\mathcal{M}_i$.[28] $\square$

F. COMPLEMENTS FOR SECTION 7

Fix an acyclic signature Σ and an artifact setting $(\underline{x}, \underline{a})$ over it. In this section we analyze in our setting the transition formulae studied in [37][29] (deletion, insertion and propagation updates). In addition, we discuss some modifications of the previous transitions and introduce new kinds of updates (like bulk updates). We prove that all these transitions are strongly local transitions.

F.1 Deletion updates

We want to remove a tuple $\underline{t} := (t_1, ..., t_m)$ from an m-ary artifact relation R and assign the values $t_1, ..., t_m$ to some of the artifact variables (let $\underline{x} := \underline{x}_1, \underline{x}_2$, where $\underline{x}_1 := (x_{i_1}, ..., x_{i_m})$ are the variables where we want to transfer the tuple $\underline{t}$). This operation has to be applied only if the current artifact variables $\underline{x}$ satisfy the pre-condition $\pi(\underline{x}_1, \underline{x}_2)$ and the updated artifact variables $\underline{x}' := \underline{x}'_1, \underline{x}'_2$ satisfy the post-condition $\psi(\underline{x}'_1, \underline{x}'_2)$ (π and ψ are quantifier-free formulae). The variables $\underline{x}_2$ are not propagated, i.e. they are non deterministically reassigned. Let $\underline{r} := r_1, ..., r_m$ be the artifact components of R. Such an update can be formalized in a symbolic way as follows:

$$\exists \underline{d}\, \exists e \left(\begin{array}{c} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{x}'_1, \underline{x}'_2) \ \wedge r_1[e] \neq \texttt{undef} \wedge ... \\ \wedge\, r_n[e] \neq \texttt{undef} \wedge (\underline{x}'_1 := \underline{r}[e] \ \wedge \ \underline{x}'_2 := \underline{d} \wedge \underline{s}' := \underline{s} \wedge \\ \wedge\, \underline{r}' := \lambda j.(\texttt{if } j = e \texttt{ then undef else } \underline{r}[j])) \end{array} \right) \tag{28}$$

where $\underline{s}$ are the artifact components of the artifact relations different from R. Notice that the $\underline{d}$ are non deterministically produced values for the updated $\underline{x}'_2$. In the terminology of [37], notice that no artifact variable is propagated in a deletion update.

Notice that in place of the condition $r_1[e] \neq \texttt{undef} \wedge ... \wedge \ r_n[e] \neq \texttt{undef}$ one can consider the modified deletion update that is fired only if *some* (and not all) artifact components are not `undef`, or even the case when the transition is fired if *at least one* artifact component is not `undef`: the latter case can be expressed using a disjunction of transitions τ_i that, instead of $r_1[e] \neq \texttt{undef} \wedge ... \wedge\ r_n[e] \neq \texttt{undef}$, involve only the literal $r_i[e] \neq \texttt{undef}$ (for $i = 1, ..., n$). These modified deletion updates can be proved to be strongly local transitions by using trivial adaptations of the arguments shown below.

The formula (28) is not in the format (6) but can be easily converted into it as follows:

$$\exists \underline{d}\, \exists e \left(\begin{array}{c} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{r}[e], \underline{d}) \ \wedge \ r_1[e] \neq \texttt{undef} \ \wedge ... \\ \wedge\, r_n[e] \neq \texttt{undef} \ \wedge \ (\underline{x}'_1 := \underline{r}[e] \ \wedge \ \underline{x}'_2 := \underline{d} \ \wedge \ \underline{s}' := \underline{s} \wedge \\ \wedge\, \underline{r}' := \lambda j.(\texttt{if } j = e \texttt{ then undef else } \underline{r}[j])) \end{array} \right) \tag{29}$$

We prove that the preimage along (29) of a strongly local formula is strongly local. Consider a strongly local formula

$$K := \psi'(\underline{x}) \wedge \exists \underline{e} \left(\text{Diff}(\underline{e}) \wedge \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}[e_r]) \wedge \Theta \right)$$

[28] The following observation shows how to extend the proof to the case where we have constants and artifact variables. Recall that in $\tilde{\Sigma}$ the artifact variables are seen as constants, so we need to consider only the case of constants. Let $\tilde{\Sigma}^+$ be $\tilde{\Sigma}$ where each constant symbol c of sort S is replaced by a new sort S_c and a new function symbol $f_c : S_c \longrightarrow S$. Now every model $\mathcal{M}$ of $\tilde{\Sigma}$ can be transformed into a model $\mathcal{M}^+$ of $\tilde{\Sigma}^+$ by interpreting S_c as a singleton set $\{*\}$ and f_c as the map sending $*$ to $c^{\mathcal{M}}$. This transformation has the following property: $\tilde{\Sigma}$-embeddings of $\mathcal{M}$ into $\mathcal{N}$ are in bijective correspondence with $\tilde{\Sigma}^+$-embeddings of $\mathcal{M}^+$ into $\mathcal{N}^+$. Since $\tilde{\Sigma}^+$ is still tree-like and does not have constant symbols, this shows that Theorem 4 holds for $\tilde{\Sigma}$ too.

[29] For simplicity, since we are not considering hierarchical aspects, we assume that there is no input variable in the sense of [37]

where Θ is a formula involving the artifact components $\underline{s}$ (which are not updated) such that no e_r occurs in it.

REMARK 5. Notice that equality is the only predicate, so a quantifier-free formula $\phi(e, \underline{a})$ involving a single variable e must be obtained from atoms of the kind $b[e] = b'[e]$ (for $b, b' \in \underline{a}$) by applying the Boolean connectives only: this is why we usually display such a formula as $\phi(\underline{a}[e])$. In addition, since the source sorts of the different artifact relations are different, we cannot employ the same variable as argument of artifact components of different artifact relations: in other words, we cannot employ the same variable e in terms like $r_i[e]$ and $s_j[e]$, in case r_i and s_j are components of two different artifact relation R and S (because e must have either type R or type S). Thus, the quantifier-free subformula $\phi_i(\underline{a}[e_i])$ in a local formula involving only the variable e_i must be of the kind $\phi_i(\underline{r}[e_i])$, for some artifact relation R (here $\underline{r}$ are the artifact components of R). These observations will be often used in the sequel. $\triangleleft$

We compute the preimage $Pre(29, K)$

$$\exists \underline{d}\, \exists e, \underline{e}\, \exists \underline{x}'_1, \underline{x}'_2\, \exists \underline{r}' \left(\begin{array}{c} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{r}[e], \underline{d}) \ \wedge \ \psi'(\underline{x}'_1, \underline{x}'_2) \ \wedge \\ \wedge \ \underline{x}'_1 := \underline{r}[e] \ \wedge \ \underline{x}'_2 := \underline{d} \ \wedge \ \mathrm{Diff}(\underline{e}) \ \wedge \ \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}'[e_r]) \ \wedge \\ \wedge \ \underline{r}' := \lambda j.(\texttt{if } j = e \texttt{ then undef else } \underline{r}[j]) \wedge \Theta \end{array} \right)$$

which can be rewritten as a disjunction of the following formulae:

- $\exists \underline{d}\, \exists e, \underline{e} \left(\begin{array}{l} \mathrm{Diff}(\underline{e}, e) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{r}[e], \underline{d}) \ \wedge \\ \wedge \ \psi'(\underline{r}[e], \underline{d}) \ \wedge \ \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}[e_r]) \ \wedge \ \Theta \end{array} \right)$
  covering the case where e is different from all $e_j \in \underline{e}$
- $\exists \underline{d}\, \exists \underline{e} \left(\begin{array}{l} \mathrm{Diff}(\underline{e}) \ \wedge \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{r}[e_j], \underline{d}) \ \wedge \ \psi'(\underline{r}[e_j], \underline{d}) \ \wedge \\ \wedge \ \bigwedge_{e_r \in \underline{e}, e_r \neq e_j} \phi_{e_r}(\underline{r}[e_r]) \wedge \phi_{e_j}(\texttt{undef}) \wedge \Theta \end{array} \right)$
  covering the case where $e = e_j$, for some $e_j \in \underline{e}$

We can now move the existential quantifier $\exists \underline{d}$ in front of $\psi \wedge \psi'$. We eliminate the quantifiers (applying the quantifier elimination procedure for $T^\star$) from the subformula $\exists \underline{d}\, (\psi(\underline{r}[e], \underline{d}) \wedge \psi'(\underline{r}[e], \underline{d}))$ (or $\exists \underline{d}\, (\psi(\underline{r}[e], \underline{d}) \wedge \psi'(\underline{r}[e], \underline{d}))$, resp.) obtaining a formula of the kind $\theta(\underline{r}[e])$ (or $\theta(\underline{r}[e_j])$.

The final result is the disjunction of the formulae

- $\exists e, \underline{e} \left(\mathrm{Diff}(\underline{e}, e) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \theta(\underline{r}[e]) \ \wedge \ \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}[e_r]) \ \wedge \ \Theta \right)$
- $\exists \underline{d}\, \exists \underline{e} \left(\begin{array}{l} \mathrm{Diff}(\underline{e}) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \theta(\underline{r}[e_j]) \ \wedge \\ \wedge \bigwedge_{e_r \in \underline{e}, e_r \neq e_j} \phi_{e_r}(\underline{r}[e_r]) \ \wedge \ \phi_{e_j}(\texttt{undef}) \ \wedge \ \Theta \end{array} \right)$

which is a strongly local formula.

Analogous arguments show that:

(i) transitions like Formula (28), where the literals $r_1[e] \neq \texttt{undef} \ \wedge ... \wedge \ r_n[e] \neq \texttt{undef}$ are replaced with a generic constraint $\chi(\underline{r}[e])$;

(ii) transitions that remove a tuple from an artifact relation (without transferring its values to the corresponding artifact variables);

(iii) transitions that copy the the content of a tuple contained in an artifact relation to some artifact variables, non-deterministically reassigning the values of the other artifact variables;

(iv) transitions that combine **(i)** and **(iii)**

are also strongly local.

REMARK 6. Notice that deletion updates with the propagation of some artifact variables $\underline{x}_1$ (which are not allowed in [37] and in [27]) are *not* strongly local, since the preimage of a strongly local formula can produce formulae of the form $\psi(\underline{r}[e], \underline{x}_1)$. This preimage is *still* local: however, the preimage of a local state formula through a deletion update can generate formulae of the form $\psi(\underline{r}[e], \underline{r}[e'])$, with $e \neq e'$, destroying locality. Hence, the safety problem for a RAS equipped containing deletion updates with propagation in its transitions, is not guaranteed to terminate. $\triangleleft$

F.2 Insertion updates

We want to insert a tuple of values $\underline{t} := (t_1, ..., t_m)$ from the artifact variables $\underline{x}_1 := (x_{i_1}, ..., x_{i_m})$ (let $\underline{x} := \underline{x}_1, \underline{x}_2$ as above) into an m-ary artifact relation R. This operation has to be applied only if the current artifact variables $\underline{x}$ satisfy the pre-condition $\pi(\underline{x}_1, \underline{x}_2)$ and the updated artifact variables $\underline{x}' := \underline{x}'_1, \underline{x}'_2$ satisfy the post-condition $\psi(\underline{x}'_1, \underline{x}'_2)$. The variables $\underline{x}$ are all not propagated, i.e. they are non deterministically reassigned. Let $\underline{r} := r_1, ..., r_m$ be the artifact components of R. Such an update can be formalized in a symbolic way as follows:

$$\exists \underline{d}_1, \underline{d}_2\, \exists e \left(\begin{array}{c} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{x}'_1, \underline{x}'_2) \ \wedge \ \underline{r}[e] = \texttt{undef} \\ \wedge \ (\underline{x}'_1 := \underline{d}_1 \ \wedge \ \underline{x}'_2 := \underline{d}_2 \ \wedge \ \underline{s}' := \underline{s} \ \wedge \\ \wedge \ \underline{r}' := \lambda j.(\texttt{if } j = e \texttt{ then } \underline{x}_1 \texttt{ else } \underline{r}[j])) \end{array} \right) \tag{30}$$

where $\underline{s}$ are the artifact components of the artifact relations different from R. Notice that $\underline{d}_1, \underline{d}_2$ are non deterministically produced values for the updated $\underline{x}'_1, \underline{x}'_2$. In the terminology of [37], notice that no artifact variable is propagated in a insertion update. Notice that the following arguments remain the same even if $\underline{r}[e] = \texttt{undef}$ is replaced with a conjunction of *some* literals of the form $r_j[e] = \texttt{undef}$, for some $j = 1, ..., m$, or even if $\underline{r}[e] = \texttt{undef}$ is replaced with a generic constraint $\chi(\underline{r}[e])$.

In this transition, the insertion of the same content in correspondence to different entries is allowed. If we want to avoid this kind of multiple insertions, the update r' must be modified as follows:

$$\underline{r}' := \lambda j. \begin{pmatrix} \texttt{if } j = e \texttt{ then } \underline{x}_1 \texttt{ else} \\ (\texttt{if } \underline{r}[j] = \underline{x}_1 \texttt{ then undef else } \underline{r}[j]) \end{pmatrix}$$

The formula (30) is not in the format (6) but can be easily converted into it as follows:

$$\exists \underline{d}_1, \underline{d}_2 \, \exists e \begin{pmatrix} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{d}_1, \underline{d}_2) \ \wedge \ \underline{r}[e] = \texttt{undef} \\ \wedge (\underline{x}'_1 := \underline{d}_1 \ \wedge \ \underline{x}'_2 := \underline{d}_2 \ \wedge \ \underline{s}' := \underline{s} \wedge \\ \wedge \ \underline{r}' := \lambda j. (\texttt{if } j = e \texttt{ then } \underline{x}_1 \texttt{ else } \underline{r}[j])) \end{pmatrix} \tag{31}$$

We prove that the preimage along (31) of a strongly local formula is strongly local. Consider a strongly local formula

$$K := \psi'(\underline{x}) \wedge \exists \underline{e} \left(\text{Diff}(\underline{e}) \wedge \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}[e_r]) \wedge \Theta \right)$$

where Θ is a formula involving the artifact relations $\underline{s}$ (which are not updated) such that no e_r occurs in it.

We compute the preimage $Pre(31, K)$

$$\exists \underline{d}_1, \underline{d}_2 \, \exists e, \underline{e} \, \exists \underline{x}'_1, \underline{x}'_2 \, \exists \underline{r}' \begin{pmatrix} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{d}_1, \underline{d}_2) \ \wedge \ \psi'(\underline{x}'_1, \underline{x}'_2) \ \wedge \ \underline{r}[e] = \texttt{undef} \\ \wedge \, \underline{x}'_1 := \underline{d}_1 \ \wedge \ \underline{x}'_2 := \underline{d}_2 \ \wedge \ \text{Diff}(\underline{e}) \ \wedge \ \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}'[e_r]) \wedge \\ \wedge \ \underline{r}' := \lambda j. (\texttt{if } j = e_1 \texttt{ then } \underline{x}_1 \texttt{ else } \underline{r}[j]) \wedge \Theta) \end{pmatrix}$$

which can be rewritten as a disjunction of the following formulae:

- $\exists \underline{d}_1, \underline{d}_2 \, \exists e, \underline{e} \begin{pmatrix} \text{Diff}(\underline{e}, e) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{d}_1, \underline{d}_2) \ \wedge \ \psi'(\underline{d}_1, \underline{d}_2) \\ \wedge \, \underline{r}[e] = \texttt{undef} \ \wedge \ \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}[e_r]) \ \wedge \ \Theta \end{pmatrix}$
  covering the case where e is different from all $e_j \in \underline{e}$
- $\exists \underline{d}_1, \underline{d}_2 \, \exists e \underline{e} \begin{pmatrix} \text{Diff}(\underline{e}) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{d}_1, \underline{d}_2) \ \wedge \ \psi'(\underline{d}_1, \underline{d}_2) \wedge \\ \wedge \, \underline{r}[e] = \texttt{undef} \ \wedge \ \bigwedge_{e_r \in \underline{e}, e_r \neq e_j} \phi_{e_r}(\underline{r}[e_r]) \wedge \phi_{e_j}(\underline{x}_1) \ \wedge \ \Theta \end{pmatrix}$
  covering the case where $e = e_j$, for some $e_j \in \underline{e}$.

We can move the existential quantifiers $\exists \underline{d}_1, \underline{d}_2$ in front of $\psi \wedge \psi'$. We eliminate the quantifiers (applying the quantifier elimination procedure for $T^\star$) from the subformula $\exists \underline{d}_1 \underline{d}_2 \, (\psi(\underline{d}_1, \underline{d}_2) \wedge \psi'(\underline{d}_1, \underline{d}_2))$ obtaining a ground formula θ.

The final result is a disjunction of formulae fo the kind

- $\exists e, \underline{e} \, \big(\text{Diff}(\underline{e}, e) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \underline{r}[e] = \texttt{undef} \ \wedge \ \theta \ \wedge \ \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}[e_r]) \ \wedge \ \Theta \big)$
- $\exists \underline{e} \, \big(\text{Diff}(\underline{e}) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \phi_{e_j}(\underline{x}_1) \ \wedge \ \underline{r}[e] = \texttt{undef} \ \wedge \ \theta \ \wedge \bigwedge_{e_r \in \underline{e}, e_r \neq e_j} \phi_{e_r}(\underline{r}[e_r]) \ \wedge \ \Theta \big)$

which is a strongly local formula.

Analogous arguments show that transitions that insert a tuple of values $\underline{t} := (t_1, ..., t_m)$ (where the values t_j are taken from the content of the artifact variables $\underline{x}_1 := (x_{i_1}, ..., x_{i_m})$ or are *constants*) into an m-ary artifact relation R are also strongly local. Notice that the transition introduced in Example 4:

$$\exists i{:}\mathsf{appIndex} \begin{pmatrix} pState = \texttt{enabled} \wedge aState = \texttt{received} \\ \wedge \ applicant[i] = \texttt{undef} \\ \wedge \ pState' = \texttt{enabled} \wedge aState' = \texttt{undef} \wedge cId' = \texttt{undef} \\ \wedge \ appJobCat' = \lambda j. \, (\mathsf{if} \ j = i \ \mathsf{then} \ jId \ \mathsf{else} \ appJobCat[j]) \\ \wedge \ applicant' = \lambda j. \, (\mathsf{if} \ j = i \ \mathsf{then} \ uId \ \mathsf{else} \ applicant[j]) \\ \wedge \ appResp' = \lambda j. \, (\mathsf{if} \ j = i \ \mathsf{then} \ eId \ \mathsf{else} \ appResp[j]) \\ \wedge \ appScore' = \lambda j. \, (\mathsf{if} \ j = i \ \mathsf{then} \ \texttt{-1} \ \mathsf{else} \ appScore[j]) \\ \wedge \ appResult' = \lambda j. \, (\mathsf{if} \ j = i \ \mathsf{then} \ \texttt{undef} \ \mathsf{else} \ appResult[j]) \\ \wedge \ jId' = \texttt{undef} \wedge uId' = \texttt{undef} \wedge eId' = \texttt{undef} \end{pmatrix}$$

presents the described format.

We close this section with an important remark. In Appendix A.1, we have seen that to forbid the insertion at different indexes of multiple identical tuples in an artifact relation, transitions break the strong locality requirement. A way to restore locality is to simply admit such repeated insertions. Notably, if one focuses on the fragment of strongly local RAS that coincides with the model in [27, 37], it can be shown, exactly reconstructing the same line of reasoning from [27], that *verification problems (in the restricted common fragment) for artifact systems working over sets (i.e., insertions are performed over working memory without possible repetitions) and those working over multisets, are indeed equivalent.*

F.3 Propagation updates

We want to propagate a tuple $\underline{t} := (t_1, ..., t_m)$ of values contained in the artifact variables $\underline{x}_1 := (x_{i_1}, ..., x_{i_m})$ (let $\underline{x} := \underline{x}_1, \underline{x}_2$) to the corresponding updated artifact variables $\underline{x}'_1$. This operation has to be applied only if the current artifact variables $\underline{x}$ satisfy the pre-condition $\pi(\underline{x}_1, \underline{x}_2)$ and the updated artifact variables $\underline{x}' := \underline{x}'_1, \underline{x}'_2$ satisfy the post-condition $\psi(\underline{x}'_1, \underline{x}'_2)$. Notice that in this transition no update of artifact component is involved.

Such an update can be formalized in a symbolic way as follows:

$$\exists \underline{d}\, (\pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{x}'_1, \underline{x}'_2) \wedge (\underline{x}'_1 := \underline{x}_1 \ \wedge \ \underline{x}'_2 := \underline{d} \ \wedge \ \underline{s}' := \underline{s})) \tag{32}$$

where $\underline{s}$ stands for all the artifact components. Notice that the $\underline{d}$ are non deterministically produced values for the updated $\underline{x}'_2$. In the terminology of [37], notice that the artifact variables $\underline{x}_1$ are propagated.

The formula (30) is not in the format (6) but can be easily converted into it as follows:

$$\exists \underline{d}\, (\pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{x}_1, \underline{d}) \wedge (\underline{x}'_1 := \underline{x}_1 \ \wedge \ \underline{x}'_2 := \underline{d} \ \wedge \ \underline{s}' := \underline{s})) \tag{33}$$

We prove that the preimage along (33) of a strongly local formula is strongly local. Consider a strongly local formula

$$K := \psi'(\underline{x}) \wedge \exists \underline{e}\, (\mathrm{Diff}(\underline{e}) \wedge \Theta)$$

where Θ is a formula involving the all artifact relations $\underline{s}$ (which are not modified in a propagation update), such that K fits the format of (9).

We compute the preimage $Pre(32, K)$

$$\exists \underline{d}\, \exists \underline{x}'_1, \underline{x}'_2 \begin{pmatrix} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{x}_1, \underline{d}) \ \wedge \ \psi'(\underline{x}_1, \underline{x}'_2) \wedge \\ \wedge \ \underline{x}'_1 := \underline{x}_1 \ \wedge \ \underline{x}'_2 := \underline{d} \ \wedge \ \mathrm{Diff}(\underline{e}) \ \wedge \ \Theta \end{pmatrix}$$

which can be rewritten as follows:

$$\exists \underline{d}\, \exists \underline{e} \begin{pmatrix} \mathrm{Diff}(\underline{e}) \wedge \pi(\underline{x}_1, \underline{x}_2) \wedge \psi(\underline{x}_1, \underline{d}) \wedge \\ \wedge \ \psi'(\underline{x}_1, \underline{d}) \ \wedge \ \Theta \end{pmatrix}$$

We can move the existential quantifier $\exists \underline{d}$ in front of $\psi \wedge \psi'$. We eliminate the quantifiers (applying the quantifier elimination procedure for $T^\star$) from the subformula $\exists \underline{d}(\psi(\underline{x}_1, \underline{d}) \wedge \psi'(\underline{x}_1 \underline{d}))$ obtaining a formula of the kind $\theta(\underline{x}_1)$.

The final result is

$$\exists \underline{e}\, (\mathrm{Diff}(\underline{e}) \ \wedge \ \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \theta(\underline{x}_1) \ \wedge \ \Theta)$$

which is a strongly local formula.

Consider a transition that inserts constants or a non-deterministically generated new value d' (or a tuple of new values $\underline{d}'$) into an artifact component r_i (or more than one) of an m-ary artifact relation $\underline{r}$, propagating all the other components and the artifact variables $\underline{x}_1$ (with $\underline{x} := \underline{x}_1, \underline{x}_2$). Formally, this transition can be written in the following way:

$$\exists \underline{d}, d'\, \exists e \begin{pmatrix} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{x}'_1, \underline{x}'_2) \ \wedge \ \chi_1(d') \ \wedge \ \chi_2(\underline{r}[e]) \wedge \\ \wedge \ (\underline{x}'_1 := \underline{x}_1 \ \wedge \ \underline{x}'_2 := \underline{d} \ \wedge \ r'_i = \lambda j.(\mathtt{if}\ j = e\ \mathtt{then}\ d'\ \mathtt{else}\ r[j]) \ \wedge \ \underline{s}' := \underline{s}) \end{pmatrix} \tag{34}$$

where $\underline{s}$ stands for all the artifact components different from r_i, and χ_1 and χ_2 are quantifier-free formulae. Notice that the $\underline{d}$ are non deterministically produced values for the updated $\underline{x}'_2$. In the terminology of [37], notice that the artifact variables $\underline{x}_1$ are propagated.

The formula (34) is not in the format (6) but can be easily converted into it as follows:

$$\exists \underline{d}, d'\, \exists e \begin{pmatrix} \pi(\underline{x}_1, \underline{x}_2) \ \wedge \ \psi(\underline{x}_1, \underline{d}) \ \wedge \ \chi_1(d') \ \wedge \ \chi_2(\underline{r}[e]) \wedge \\ \wedge \ (\underline{x}'_1 := \underline{x}_1 \ \wedge \ \underline{x}'_2 := \underline{d} \ \wedge \ r'_i = \lambda j.(\mathtt{if}\ j = e\ \mathtt{then}\ d'\ \mathtt{else}\ r[j]) \ \wedge \ \underline{s}' := \underline{s}) \end{pmatrix} \tag{35}$$

Since d' does not occur in literals involving artifact variables, arguments analogous to the previous ones show that this transition is strongly local.

Notice that the transition (described in Example 4):

$$\exists i{:}\mathsf{jobIndex}, s{:}\mathsf{Score} \begin{pmatrix} pState = \texttt{enabled} \\ \wedge\ applicant[i] \neq \texttt{undef} \wedge appScore[i] = \texttt{-1} \wedge s \geq 0 \\ \wedge\ pState' = \texttt{enabled} \wedge appScore'[i] = s \end{pmatrix}$$

that assesses a Score to an applicant presents the structure of (35), so it is a strongly local transition. The same conclusion holds for the transition:

$$\exists u{:}\mathsf{UserId}, j{:}\mathsf{JobCatId}, e{:}\mathsf{EmpId}, c{:}\mathsf{CompInId} \begin{pmatrix} pState = \texttt{enabled} \wedge aState = \texttt{undef} \\ \wedge\ u \neq \texttt{undef} \wedge j \neq \texttt{undef} \wedge e \neq \texttt{undef} \wedge c \neq \texttt{undef} \\ \wedge\ who(c) = e \wedge what(c) = j \\ \wedge\ pState' = \texttt{enabled} \wedge aState' = \texttt{received} \\ \wedge\ uId' = u \wedge jId' = j \wedge eId' = e \wedge cId' = c \end{pmatrix}$$

presented in Example 4.

F.4 Bulk update

We want to unboundedly (bulk) update one (or more than one) artifact component(s) r_i of one (or more than one) artifact relation(s) $\underline{r}$: if some conditions over the artifacts are satisfied for some entries, a global update that involves all those entries (inserting some constant c_1) is fired. In our symbolic formalism, we write:

$$\exists \underline{d} \left(\begin{array}{c} \pi(\underline{x}_1, \underline{x}_2) \ \wedge\ \psi(\underline{x}'_1, \underline{x}'_2) \ \wedge (\underline{x}'_1 := \underline{x}_1 \ \wedge\ \underline{x}'_2 := \underline{d} \ \wedge\ \underline{s}' := \underline{s} \wedge \\ \wedge\ r'_1 := r_1 \ \wedge ... \wedge\ r'_i := \lambda j.(\texttt{if}\ \kappa_1(\underline{r}[j])\ \texttt{then}\ c_1\ \texttt{else}\ r_i[j])) \ \wedge ... \wedge\ r'_n := r_n) \end{array} \right) \tag{36}$$

where $\underline{r}$ are the artifact components of an artifact relation R, $\underline{s}$ are the remaining artifact components, κ_1 is a quantifier-free formula[30], c_1 is a constant. The artifact component r_i is updated in a global, unbounded way: we call this kind of update "bulk update".

The formula (36) is not in the format (6) but can be easily converted into it as follows:

$$\exists \underline{d} \left(\begin{array}{c} \pi(\underline{x}_1, \underline{x}_2) \ \wedge\ \psi(\underline{x}_1, \underline{d}) \ \wedge (\underline{x}'_1 := \underline{x}_1 \ \wedge\ \underline{x}'_2 := \underline{d} \ \wedge\ \underline{s}' := \underline{s} \wedge \\ \wedge\ r'_1 := r_1 \ \wedge ... \wedge\ r'_i := \lambda j.(\texttt{if}\ \kappa_1(\underline{r}[j])\ \texttt{then}\ c_1\ \texttt{else}\ r_i[j])) \ \wedge ... \wedge\ r'_n := r_n) \end{array} \right) \tag{37}$$

We prove that the preimage along (37) of a strongly local formula is strongly local. Consider a strongly local formula

$$K := \psi'(\underline{x}) \wedge \exists \underline{e} \left(\text{Diff}(\underline{e}) \wedge \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}[e_r]) \wedge \Theta \right)$$

where Θ is a formula involving the artifact relations $\underline{s}$ (which are not updated) such that no e_r occurs in it.

We compute the preimage $Pre(37, K)$

$$\exists \underline{d}\, \exists \underline{e} \left(\begin{array}{c} \text{Diff}(\underline{e}) \ \wedge\ \pi(\underline{x}_1, \underline{x}_2) \ \wedge\ \psi(\underline{x}_1, \underline{d}) \ \wedge \psi'(\underline{x}_1, \underline{d}) \ \wedge (\underline{x}'_1 := \underline{x}_1 \ \wedge\ \underline{x}'_2 := \underline{d} \ \wedge\ \underline{s}' := \underline{s} \wedge \\ \bigwedge_{e_r \in \underline{e}} \phi_{e_r}(\underline{r}'[e_r]) \ \wedge\ \Theta \ \wedge\ r'_1 := r_1 \ \wedge ... \wedge\ r'_i := \lambda j.(\texttt{if}\ \kappa_1(\underline{r}[j])\ \texttt{then}\ c_1\ \texttt{else}\ r_i[j])) \ \wedge ... \wedge\ r'_n := r_n) \end{array} \right) \tag{38}$$

which can be rewritten as a disjunction of the following formulae indexed by a function f that associates to every e_r a boolean value in $0, 1$:

$$\exists \underline{d}, \exists \underline{e} \left(\begin{array}{c} \text{Diff}(\underline{e}) \ \wedge\ \pi(\underline{x}_1, \underline{x}_2) \ \wedge\ \psi(\underline{x}_1, \underline{d}) \ \wedge \psi'(\underline{x}_1, \underline{d}) \wedge \\ \bigwedge_{e_r \in \underline{e}} (\epsilon_f(e_r) \kappa_1(\underline{r}[e_r]) \ \wedge\ \phi(r_1[e_r], ... \delta_f(e_r), ..., r_n[e_r])) \ \wedge\ \Theta \end{array} \right) \tag{39}$$

where $\epsilon_f(e_r) := \neg$ if $f(e_r) = 0$, otherwise $\epsilon_f(e_r) := \emptyset$, and $\delta_f(e_r) := c_1$ if $f(e_r) = 0$, otherwise $\delta_f(e_r) := r_i[e_r]$.

We can conclude as above (cf. propagation updates), by eliminating the existentially quantified variable $\underline{d}$, that this formula is strongly local.

Notice that the previous arguments remain the same if $r'_i := \lambda j.(\texttt{if}\ \kappa_1(\underline{r}[j])\ \texttt{then}\ c_1\ \texttt{else}\ r_i[j]))$ in Formula (36) is replaced by $r'_i := \lambda j.(\texttt{if}\ \kappa_1(\underline{r}[j])\ \texttt{then}\ c_1\ \texttt{else}\ c_2)$, with c_2 a constant. Even in this case, the modified bulk transition is strongly local.

[30] From the computations below, it is clear that strong locality holds also in case κ_1 depends also on the variables $\underline{x}$, on the condition that $\kappa_1(\underline{x}, \underline{r}[j])$ has the form $h_0(\underline{x}) \wedge h_1(\underline{r}[j])$, with h_0 and h_1 quantifier-free formulae

	Example	#(AC)	#(AV)	#(T)
E1	JobHiring	9	18	15
E2	Acquisition-following-RFQ	6	13	28
E3	Book-Writing-and-Publishing	4	14	13
E4	Customer-Quotation-Request	9	11	21
E5	Patient-Treatment-Collaboration	6	17	34
E6	Property-and-Casualty-Insurance-Claim-Processing	2	7	15
E7	Amazon-Fulfillment	2	28	38
E8	Incident-Management-as-Collaboration	3	20	19

Table 2: Summary of the experimental examples

Example	Property	Result	Time	#(N)	depth	#(SMT-calls)
E1	E1P1	SAFE	0.06	3	3	1238
	E1P2	UNSAFE	0.36	46	10	2371
	E1P3	UNSAFE	0.50	62	11	2867
	E1P4	UNSAFE	0.35	42	10	2237
E2	E2P1	SAFE	0.72	50	9	3156
	E2P2	UNSAFE	0.88	87	10	4238
	E2P3	UNSAFE	1.01	92	9	4811
	E2P4	UNSAFE	0.83	80	9	4254
E3	E3P1	SAFE	0.05	1	1	700
	E3P2	UNSAFE	0.06	14	3	899
E4	E4P1	SAFE	0.12	14	6	1460
	E4P2	UNSAFE	0.13	18	8	1525
E5	E5P1	SAFE	4.11	57	9	5618
	E5P2	UNSAFE	0.17	13	3	2806
E6	E6P1	SAFE	0.04	7	4	512
	E6P2	UNSAFE	0.08	28	10	902
E7	E7P1	SAFE	1.00	43	7	5281
	E7P2	UNSAFE	0.20	7	4	3412
E8	E8P1	SAFE	0.70	77	11	3720
	E8P2	UNSAFE	0.15	25	7	1652

Table 3: Experimental results for safety properties

Analogous arguments show that transitions involving more than one artifact relations which are updated like r_i are also strongly local.

The transition introduced in Example 4

$$pState = \texttt{final} \wedge pState' = \texttt{notified}$$
$$\wedge\ appResult' = \lambda j.\ \begin{pmatrix}\textsf{if } appScore[j] > 80 \textsf{ then } \texttt{winner}\\ \textsf{else } \texttt{loser}\end{pmatrix}$$

is a bulk update transition in the format described in this subsection, so it is a strongly local transition.

G. EXPERIMENTS

We base our experimental evaluation on the already existing benchmark provided in [37], that samples 32 real-world BPMN workflows published at the official BPM website (`http://www.bpmn.org/`). Specifically, inspired by the specification approach adopted by the authors of [37] in their experimental setup (`https://github.com/oi02lyl/has-verifier`), we select seven examples of varying complexity (see Table 2) and provide their faithful encoding[31] in the array-based specification using MCMT version 2.8 (`http://users.mat.unimi.it/users/ghilardi/mcmt/`). Moreover, we enrich our experimental set with an extended version of the running example from Appendix A.1. Each example has been checked against at least one safe and one unsafe conditions. Experiments were performed on a machine with Ubuntu 16.04, 2.6 GHz Intel Core i7 and 16 GB RAM.

[31] Our encoding considers semantics of the framework studied in [37].

Here **#(AV)**, **#(AC)** and **#(T)** represent, respectively, the number of artifact variables, artifact components and transitions used in the example specification, while **Time** is the MCMT execution time. The most critical measures are **#(N)**, **depth** and **#(SMT-calls)** that respectively define the number of nodes and the depth of the tree used for the backward reachability procedure adopted by MCMT, and the number of the SMT-solver calls. Indeed, MCMT computes the iterated preimages of the formula describing the unsafe states along the various transitions. Such computation produces a tree, whose nodes are labelled by formulae describing sets of states that can reach an unsafe state and whose arcs are labelled by a transition. In other words, an arc $t : \phi \rightarrow \psi$ means that ϕ is equal to $Pre(t, \psi)$. The tool applies forward and backward simplification strategies, so that whenever a node ϕ is deleted, this means that ϕ entails the disjunction of the remaining (non deleted) nodes. All nodes (both deleted and undeleted) can be visualized via the available online options (it is also possible to produce a Latex file containing their detailed description)

To stress test our encoding, we came up with a few formulae describing unsafe configurations (sets of "bad" states), that is, the configurations that the system should not incur throughout its execution. **Property** references encodings of examples endowed with specific (un)safety properties done in MCMT, whereas **Result** shows their verification outcome that can be of the two following types: `SAFE` and `UNSAFE`. The MCMT tool returns `SAFE`, if the undesirable property it was asked to verify represents a configuration that the system cannot reach. At the same time, the result is `UNSAFE` if there exists a path of the system execution that reaches "bad" states. One can see, for example, that the job hiring RAS has been proved by MCMT to be `SAFE` w.r.t. the property defined in Example 5. The details about the successfully completed verification task can be seen in the first row of Table 3: the tool constructed a tree with 3 nodes and a depth of 3, and returned `SAFE` in 0.06 seconds. For the same job hiring RAS, if we slightly modify the safe condition discussed in Example 5 by removing, for instance, the check that a selected applicant is not a winning one, we obtain a description (see below) of a configuration in which it is still the case that an applicant could win:

$$\exists i{:}\mathsf{appIndex}\,(pState = \texttt{notified} \wedge applicant[i] \neq \texttt{undef} \wedge appResult[i] \neq \texttt{loser})$$

In this case, the job hiring process analyzed against the devised property is evaluated as `UNSAFE` by the tool (see E1P3 row in Table 3). When checking safety properties, MCMT also allows to access an unsafe path of a given example in case the verification result is `UNSAFE`.

To conclude, we would like to point out that seemingly high number of SMT solver calls in **#(SMT-calls)** against relatively small execution time demonstrates that MCMT could be considered as a promising tool supporting the presented line of research. This is due to the following two reasons. On the one hand, the SMT technology underlying solvers like YICES [29] is quite mature and impressively well-performing. On the other hand, the backward reachability algorithm generates proof obligations which are relatively easy to be analyzed as (un)satisfiable by the solver.