From Model Completeness to Verification of Data Aware Processes

Diego Calvanese[1], Silvio Ghilardi[2], Alessandro Gianola[1(✉)], Marco Montali[1], and Andrey Rivkin[1]

[1] Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy
{calvanese,gianola,montali,rivkin}@inf.unibz.it

[2] Dipartimento di Matematica, Università degli Studi di Milano, Milan, Italy
silvio.ghilardi@unimi.it

Abstract. Model Completeness is a classical topic in model-theoretic algebra, and its inspiration sources are areas like algebraic geometry and field theory. Yet, recently, there have been remarkable applications in computer science: these applications range from combined decision procedures for satisfiability and interpolation, to connections between temporal logic and monadic second order logic and to model-checking. In this paper we mostly concentrate on the last one: we study verification over a general model of so-called artifact-centric systems, which are used to capture business processes by giving equal important to the control-flow and data-related aspects. In particular, we are interested in assessing (parameterized) safety properties irrespectively of the initial database instance. We view such artifact systems as array-based systems, establishing a correspondence with model checking based on Satisfiability-Modulo-Theories (SMT). Model completeness comes into the picture in this framework by supplying quantifier elimination algorithms for suitable existentially closed structures. Such algorithms, whose complexity is unexpectedly low in some cases of our interest, are exploited during search and to represent the sets of reachable states. Our first implementation, built up on top of the MCMT model-checker, makes all our foundational results fully operational and quite effective, as demonstrated by our first experiments.

1 Introduction

In this introduction, we briefly review some results coming from joint work of Franz Baader with the second author during the years 2004–2012: the novel contributions of the present paper can in fact be considered as a natural continuation of such previous cooperation. In both cases, the common background is the attempt of reinterpreting a classical model-theoretic tool (namely model-completeness) inside the realm of computational logic and of automated reasoning. In former joint work the focus was related to the combination of decision

https://doi.org/10.1007/978-3-030-22102-7_10

procedures in first order theories, in the present paper the focus is tailored to the use of decision procedures in declarative model-checking (in particular, in model-checking oriented to the emerging area of verification of data aware processes).

Finding solutions to equations is a challenge at the heart of both mathematics and computer science. Model-theoretic algebra, originating with the groundbreaking work of Robinson [55,56], cast the problem of solving equations in a logical form, and used this setting to solve algebraic problems via model theory. The central notion is that of an *existentially closed model*, which we explain now. Call a quantifier-free formula with parameters in a model $\mathcal{M}$ *solvable* if there is an extension $\mathcal{M}'$ of $\mathcal{M}$ where the formula is satisfied. A model $\mathcal{M}$ is *existentially closed* if any solvable quantifier-free formula already has a solution in $\mathcal{M}$ itself. For example, the field of real numbers is not existentially closed, but the field of complex numbers is.

Although this definition is formally clear, it has a main drawback: it is not first-order definable in general. However, in fortunate and important cases, the class of existentially closed models of a first-order theory T are exactly the models of another first-order theory T^*. In this case, the theory T^* can be characterized abstractly as the *model companion* of T. Model companions become *model completions* (cf. Definition 2.1) in the case of universal theories with the amalgamation property; in such model completions, quantifier elimination holds, unlike in the original theory T. The model companion/model completion of a theory identifies the class of those models where *all satisfiable existential statements can be satisfied.* For example, the theory of algebraically closed fields is the model companion of the theory of fields, and dense linear orders without endpoints give the model companion of linear orders.

1.1 Model Completeness in Combined Decision Problems

A first application of model completeness in computer science, more specifically in automated reasoning, was related to the area of *Satisfiability Modulo Theories* (SMT). The SMT-LIB project[1] (started in 2003) aims at bringing together people interested in developing powerful tools combining sophisticated techniques in SAT-solving with dedicated decision procedures involving specific theories used in applications (especially in software verification).

One of the main problems in the SMT area is to design algorithms for *constraint satifiability problems modulo a given theory T*: in such problems, one is given a finite set of literals and is asked to determine whether this set is satisfiable in a model of T. Theories of interests include linear (real and integer) arithmetics and its fragments, as well as theories axiomatizing datatypes like lists, arrays, etc. Very often such theories come out as *combination* of one or more component theories (arrays of integers, reals, booleans are typical examples) and one would like to obtain constraint satisfiability algorithms for combined theories in a *modular way.* The simplest way to implement this is to have a specific module for

[1] http://smtlib.cs.uiowa.edu/.

each component theory and to leave such modules to exchange information concerning the clauses expressible in the shared signature. This simple methodology is quite attractive, but unfortunately not complete in general. A sufficient condition for completeness was identified in [37]: the exchange procedure is complete in case the theory axiomatizing the shared signature reduct T_0 has a model completion T_0^* and each of the component theories T_i is T_0-compatible, i.e., every model of T_i embeds into a model of $T_0^* \cup T_i$. Intuitively, the reason why this condition is sufficient is the fact that one can check satisfiability of constraints in the combined signature by restricting to models whose reduct to the shared signature is a model of T_0^*, so that quantifier elimination in T_0^* guarantees that exchanging information over the quantifier-free fragment is sufficient. This result from [37] generalizes to the non-disjoint signatures case the well-known Nelson-Oppen method [51,61], because to be stably infinite in the sense of [51] means precisely to be compatible with the pure equality theory. For the above outlined exchange procedure to yield decidability of the combined constraint satisfiability problem, we need (for termination) a further hypothesis, namely that the shared theory T_0 is locally finite (which means that the total amount of information that needs to be exchanged is finite up to T_0-equivalence). All the above hypotheses apply for instance to the case of modal algebras with operators, yieldying as a by-product the well-known fusion transfer result for decidability of the global consequence relation in modal logic [63].

The results from [37] however do not supply a sufficient condition for decidability of *combined word problems*. The case of combined word problems is in a sense more challenging: we assume that the component algorithms are only able to test (un)satifiability of a single disequation and we want to conclude that the same property can be transferred to the combined theory via suitable information exchange. In the disjoint signature case, combined word problems are always decidable in case the component equational theories have a decidable word problem [52]; however, for non-disjoint signatures, combined algorithms were known only in case the component theories satisfy a kind of term factorization property [14,36]. In [12], it was proved that T_0-compatibility, joined with a special Gaussian property, yields also here a combined decidability result; the result has again a remarkable consequence in modal logic, as it implies the fusion transfer result for decidability of the *local* consequence relation (this solves a long-standing open question and, up to now and as far as we know, the proof supplied in [12] via general combination methods is the only available proof of this result).

The above methodology was further extended to cover different combination schemata for first order theories [9–11] (again having as special case a combination schema, namely E-connections [49], introduced in the framework of modal and description logics).

Model completeness has further application in automated reasoning: it has been applied to design complete algorithms for constraint satisfiability in theory extensions [59,60] and for combination transfer for quantifier-free interpolation (both for modal logics and for software verification theories) [38,39]. Another different research line used model completeness in order to discover interesting connections between monadic second order logic and its temporal logic fragments [43,44].

1.2 Towards Model Completeness in Verification

In order to see the connection between model completeness and verification, the following simple but nevertheless important observation is crucial. In declarative approaches to model-checking, the runs of a system are identified with certain definable paths in the models of a theory T: in case transition systems are represented via quantifier-free formulae and system variables are modeled as first order variables, it is easy to see that, without loss of generality and as far as safety problems are concerned, one may *restrict to paths within existentially closed models*, thus taking profit from the properties enjoyed by the model completion T^* of T whenever it exists. In particular, during forward or backward search, one can exploit quantifier elimination in order to represent sets of reachable states via quantifier-free formulae.[2]

Our intended applications are however more complex, because we need to handle transition systems whose variables are not just individual first order variables. The systems we have in mind are generically called *array-based systems*, where the term "array-based systems" is an umbrella term generically referring to infinite-state transition systems implicitly specified using a declarative, logic-based formalism comprising *second order function variables*. The formalism captures transitions manipulating arrays via logical formulae, and its precise definition depends on the specific application of interest. The first declarative formalism for array-based systems was introduced in [40,41] to handle the verification of distributed systems, and afterwards was successfully employed also to verify a wide range of infinite-state systems [4,8]. 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 function variables: they map *indexes* to *data*, in a way that changes as the system evolves.

Quantifiers are then used to represent sets of system states, however the kind of formulae that are needed for this purpose obey specific syntactic restrictions. Due to these restrictions, the proof obligations generated during model checking search (usually *backward search* is implemented in these systems) can

[2] It is quite curious to notice that this observation (in its essence) was already present in the paper [45], where however model completeness was not mentioned at all! Instead of quantifier elimination in the model completion T^*, the authors of [45] relied on the computation of the so called 'cover' of an existential formula (such cover turns out to be equivalent to the quantifier free equivalent formula modulo T^*).

be discharged by techniques combining *instantiation algorithms* with *quantifier elimination algorithms*. Typically, quantifiers ranging over indexes are handled by instantiation and quantifiers over data are handled via quantifier elimination (whenever quantifier elimination is considered too expensive or whenever there is the need to speed up termination, other techniques like interpolation or abstraction may be preferred to quantifier elimination).

The above discussion makes the step we are planning to make in the following evident: whenever quantifier elimination for data is not available, one may *resort to model completions to handle data quantifiers* arising during search in array-bases systems. This is not an abstract plan in fact, because there is an emerging area in verification that leads precisely to this, namely the area of verification of data aware processes (see below). We just mention another crucial fact from the implementation point of view: the cost of quantifier elimination in the model completions relevant for the application area of data aware processes is surprisingly low. In fact, eliminating a tuple of quantified variables from a primitive formula requires only polynomial time and can be achieved for instance via ground Knuth-Bendix completion, see [23] for more details.[3]

1.3 Data Aware Processes: Our Contribution

During the last two decades, a huge body of research has been dedicated to the challenging problem of reconciling data and process management within contemporary organizations [33,53,54]. 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. Striving for this integration, new models were devised, with two prominent representatives: object-centric processes [48], and business artifacts [29,46].

In parallel, a flourishing series of results has been dedicated to the formalization of such integrated models, and to the boundaries of decidability and complexity for their static analysis and verification [20]. Such results are quite fragmented, since they consider a variety of different assumptions on the model and on the static analysis tasks [20,62]. 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 (DB) with arbitrary first-order constraints [1,15,16,21]. Actions amount to full bulk updates that may simultaneously operate on multiple tuples, possibly injecting fresh values taken from an infinite data domain. Verification is studied by fixing the initial instance of the DB, and by considering all possible evolutions induced by the process over the initial data.

A second trend of research is instead focused on the formalization and verification of artifact-centric processes. These systems are traditionally formalized using three components [28,31]: *(i)* a read-only DB that stores fixed, background

[3] Again, without mentioning any specific application, this was already observed in [45], as the specialization of the cover algorithm to signatures with unary free function symbols.

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, obtained via a careful tuning of the relative expressive power of its three components, have been studied towards decidability of verification problems parameterized over the read-only DB (see, e.g., [17,28,31,32]). These are verification problems where a property is checked for every possible configuration of the read-only DB. For instance, for the working memory, radically different models are obtained depending on whether only a single artifact instance is evolved, or whether instead the co-evolution 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 one to capture the evolution of a single artifact instance [31]. We call an artifact system of this form *Simple Artifact System (SAS)*. Instead, more sophisticated types of artifact systems have been studied recently in [32,50]. 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 evolved on its own. We call an artifact system of this form *Relational Artifact System (RAS)*.

The overarching goal of this work is to connect, for the first time, such formal models and their corresponding verification problems, with the models and techniques of *model checking via array-based systems* described above. This is concretized through four technical contributions.

Our *first contribution* is the definition of *a general framework of so-called RASs*, in which artifacts are formalized in the spirit of array-based systems. In this setting, SASs are a particular class of RASs, where only artifact variables are allowed. RASs employ arrays to capture a very rich working memory that simultaneously accounts for artifact variables storing single data elements, and for full-fledged artifact relations storing unboundedly many tuples. Each artifact relation is captured using a collection of arrays, so that a tuple in the relation can be retrieved by inspecting the content of the arrays with a given index. The elements stored therein may be fresh values injected into the RAS, or data elements extracted from the read-only DB, whose relations are subject to key and foreign key constraints. This constitutes a big leap from the usual applications of array-based systems, because the nature of such constraints is quite different and requires completely new techniques for handling them (for instance, for quantifier elimination, as mentioned above). To attack this complexity, by relying on array-based systems, RASs encode the read-only DB using a functional, algebraic view, where relations and constraints are captured using multiple sorts and unary functions. The resulting model captures the essential aspects of the model in [50], which in turn is tightly related (though incomparable) to the sophisticated formal model for artifact-centric systems of [32].

Our *second contribution* is the development of *algorithmic techniques* for the verification of *(parameterized) safety* properties over RASs. This amounts to determining whether there exists an instance of the read-only DB that allows the RAS to evolve from its initial configuration to an *undesired* one that falsifies a given state property. To attack this problem, we build on backward reachability search [40,41]. This is a correct, possibly non-terminating technique that *regresses* the system from the undesired configuration to those configurations that reach the undesired one. This is done by iteratively computing symbolic pre-images, until they either intersect the initial configuration of the system (witnessing unsafety), or they form a fixpoint that does not contain the initial state (witnessing safety).

Adapting backward reachability to the case of RASs, by retaining soundness and completeness, requires genuinely novel research so as to eliminate new (existentially quantified) "data" variables introduced during regression. Traditionally, this is done by quantifier instantiation or elimination. However, while quantifier instantiation can be transposed to RASs, quantifier elimination cannot, since the data elements contained in the arrays point to the content of a full-fledged DB with constraints. To reconstruct quantifier elimination in this setting, which is the main technical contribution of this work, we employ the classic model-theoretic machinery of model completions: via model completions, we prove that the runs of a RAS can be faithfully lifted to richer contexts where quantifier elimination is indeed available, despite the fact that it was not available in the original structures. This allows us to recast safety problems over RASs into equivalent safety problems in this richer setting.

Our *third contribution* is the identification of *three notable classes of RASs* for which backward reachability terminates, in turn witnessing decidability of safety. The first class restricts the working memory to variables only, i.e., focuses on SASs. The second class focuses on RASs operating under the restrictions imposed in [50]: it requires acyclicity of foreign keys and ensures a sort of locality principle where different artifact tuples are not compared. Consequently, it reconstructs the decidability result exploited in [50] if one restricts the verification logic used there to safety properties only. In addition, our second class supports full-fledged bulk updates, which greatly increase the expressive power of dynamic systems [57] and, in our setting, witness the incomparability of our results and those in [50]. The third class is genuinely novel, and while it further restricts foreign keys to form a tree-shaped structure, it does not impose any restriction on the shape of updates, and consequently supports not only bulk updates, but also comparisons between artifact tuples.

Our *fourth contribution* concerns the *implementation of backward reachability techniques* for RASs. Specifically, we have extended the well-known MCMT model checker for array-based systems [42], obtaining a fully operational counterpart to all the foundational results presented in the paper. Even though implementation and experimental evaluation are not central in this paper, we note that our model checker correctly handles the examples produced to test VERIFAS [50], as well as additional examples that go beyond the verification capabilities of VERIFAS, and report some interesting cases 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.

This paper is essentially a survey and is meant to summarize ongoing work (cf. [22]); results are stated without proofs or with just proof sketches (proofs are all available in the extended version [24]). The rest of the paper is structured as follows. We give necessary preliminaries in Sect. 2. We present our functional view of (read-only) DBs with constraints in Sect. 3, and we introduce the RAS formal model in Sect. 4. We study safety via backward reachability in Sect. 5, and termination of backward reachability in Sect. 6. We report on our implementation effort and related experiments in Sect. 7, and conclude the paper in Sect. 8.

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 with constants and function symbols f, each of which has *sources* $\underline{S}$ and a *target* S', denoted as $f : \underline{S} \longrightarrow S'$; similarly relation simbols R have sources, written as $R : \underline{S}$. 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* if there is a model $\mathcal{M}$ of T and an assignment to the free variables of ϕ that make ϕ true in $\mathcal{M}$.

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 $\underline{x}$ such that $\mathcal{M}, \alpha \models \exists \underline{y}\, \phi(\underline{x}, \underline{y})$?

A theory T has *quantifier elimination* iff for every formula $\phi(\underline{x})$ in the signature of T there is a quantifier-free formula $\phi'(\underline{x})$ such that $T \models \phi(\underline{x}) \leftrightarrow \phi'(\underline{x})$. It is well-known (and easily seen) that quantifier elimination holds in case we can eliminate quantifiers from *primitive* formulae, i.e., from formulae of the kind $\exists \underline{y}\, \phi(\underline{x}, \underline{y})$, where ϕ is a constraint. Since we are interested in effective computability, we assume that *whenever* we talk about quantifier elimination, an *effective procedure* for eliminating quantifiers is given.

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

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 $\Sigma^{\underline{a}}$. 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 $\mathcal{A}' := (\mathcal{A}, a)_{a \in |\mathcal{A}|}$ by just interpreting the additional constants 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 $\mathcal{A}$ 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 $(\mathcal{M} \models \varphi \;\Rightarrow\; \mathcal{N} \models \varphi)$ 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 the last condition 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*.

In the following (cf. Sect. 4), we specify transitions of an artifact-centric system using first-order formulae. To obtain a more compact representation, we make use there of definable extensions as a means for introducing so-called *case-defined functions*. We fix a signature Σ and a Σ-theory T; a *T-partition* is a finite set $\kappa_1(\underline{x}), \dots, \kappa_n(\underline{x})$ 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 $\kappa_1(\underline{x}), \dots, \kappa_n(\underline{x})$ together with Σ-terms $t_1(\underline{x}), \dots, t_n(\underline{x})$ (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., $F \notin \Sigma$)[5], 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 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 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 (3) below) for a more compact (still first-order) 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).

We recall a standard notion in Model Theory, namely the notion of a *model completion* of a first order theory [26] (we limit the definition to universal theories, because we shall use only this case):

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

Definition 2.1. *Let T be a universal Σ-theory and let $T^\star \supseteq T$ be a further Σ-theory; we say that $T^\star$ is a model completion of T iff: (i) every model of T can be embedded into a model of $T^\star$; (ii) for every model $\mathcal{M}$ of T, we have that $T^\star \cup \Delta_\Sigma(\mathcal{M})$ is a complete theory in the signature $\Sigma^{|\mathcal{M}|}$.*

Since T is universal, condition *(ii)* is equivalent to the fact that $T^\star$ *has quantifier elimination*; on the other hand, a standard argument (based on diagrams and compactness) shows that condition *(i)* is the same as asking that T and $T^\star$ have the same universal consequences. Thus we have an equivalent definition (to be used in the following):

Proposition 2.2. *Let T be a universal Σ-theory and let $T^\star \supseteq T$ be a further Σ-theory; $T^\star$ is a model completion of T iff: (i) every Σ-constraint satisfiable in a model of T is also satisfiable in a model of T^*; (ii) T^* has quantifier elimination.*

We recall also that the model completion $T^\star$ of a theory T is unique, if it exists (see [26] for these results and for examples).

3 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.

Definition 3.1. *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 relation symbols (of any arity), equality, unary function symbols, and constants; (ii) T is a* DB theory, *that is, a set of universal Σ-sentences.*

Relation symbols are used to represent plain relations, whereas unary function symbols are used to represent relations endowed with primary and foreign key constraints (as will be explained in Sect. 3.1 below). We refer to a DB schema simply through its (DB) signature Σ and (DB) theory T, and denote by Σ_{srt} the set of sorts, by Σ_{rel} the set of relations, and by Σ_{fun} the set of functions in Σ. Since Σ contains only unary function symbols and equality, each atomic Σ-formula is of the form $t_1(v_1) = t_2(v_2)$ or $R(t_1(v_1), \ldots, t_n(v_n))$, where $t_1, t_2, \ldots, t_n$ are possibly complex terms, and $v_1, v_2, \ldots, v_n$ are 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 into 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 3.2. *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.*

Contrast this to arbitrary *models* of T, where no finiteness assumption is made. What may appear as not customary in Definition 3.2 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 Sect. 3.1.

We respectively denote by $S^{\mathcal{M}}$, $R^{\mathcal{M}}$, $f^{\mathcal{M}}$, and $c^{\mathcal{M}}$ the interpretation in $\mathcal{M}$ of the sort S (this is a set), of the relation symbol R (this is a set of tuples), 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, $f^{\mathcal{M}}$, $R^{\mathcal{M}}$, 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 $U^{\mathcal{M}}$.

Example 3.3. 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 potential job applicants; *(ii)* the different, available job categories; *(iii)* employees belonging to HR, together with the job categories they are competent in. To formalize these different aspects, we make use of a DB signature Σ_{hr} consisting of: *(i)* four id sorts UserId, EmpId, $\mathsf{CompInId}$, and $\mathsf{JobCatId}$, used to respectively identify users, employees, job categories, and the competence relationship connecting employees to job categories; *(ii)* one value sort String, containing strings used to name users and employees, and to describe job categories; and *(iii)* five function symbols, namely: *userName* and *empName*, respectively

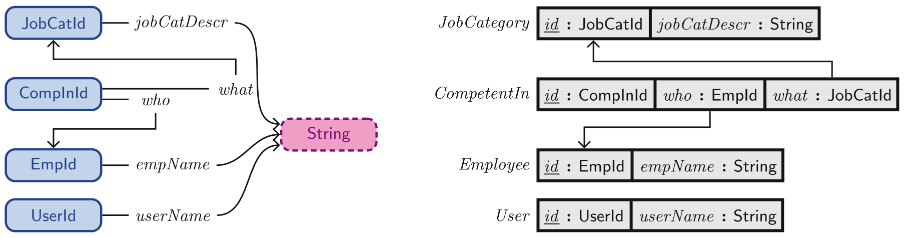

Fig. 1. On the left: characteristic graph of the human resources DB signature from Example 3.3. On the right: relational view of the DB signature; each cell denotes an attribute with its type, underlined attributes denote primary keys, and directed edges capture foreign keys.

mapping user identifiers and employee identifiers to their corresponding names; *jobCatDescr*, mapping job category identifiers to their corresponding descriptions; and *who* and *what*, mapping competence identifiers to their corresponding employees and job categories, respectively. The characteristic graph of Σ_{hr} is shown in the left part of Fig. 1. $\triangleleft$

We close the formalization of DB schemas by discussing DB theories, whose role is to encode background axioms. We illustrate a typical background axiom, required to handle the possible presence of *undefined identifiers/values* in the different sorts. This axiom is essential to capture artifact systems whose working memory is initially undefined, in the style of [32,50]. 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:

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

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 3.4. In the artifact-centric model in the style of [32,50] that we intend to capture, the DB theory consists of Axioms (1) only. However, our technical results do not require this specific choice, and more general sufficient conditions will be discussed later. These conditions apply to natural variants of Axiom (1) (such variants might be used to model situations where we would like to have, for instance, many undefined values, see [24]).

3.1 Relational View of DB Schemas

We now clarify how the algebraic, functional characterization of DB schemas and instances can be actually reinterpreted in the classical, relational model. Definition 3.1 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 (only for this subsection, we assume that Σ_{rel} is empty, for simplicity, because we want to concentrate on the most sophisticated part of our formal model, the part aiming at formalizing key dependencies). 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 3.5. The diagram on the right in Fig. 1 graphically depicts the relational view corresponding to the DB signature of Example 3.3. ◁

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

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

consisting of all (proper) identifiers assigned by $\mathcal{M}$ to id sorts, as well as all values obtained in $\mathcal{M}$ via the application of some function. Since such values are necessarily *finitely many*, one may wonder why in Definition 3.2 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. From the definition of active domain above, exploiting Axioms (1) we get that the membership of a tuple $(x_0, \ldots, x_n)$ to a generic $n+1$-ary relation R_S with key dependencies (corresponding to an id sort S) can be expressed in our setting by using just n unary function symbols and equality:

$$R_S(x_0, \ldots, x_n) \quad \text{iff} \quad x_0 \neq \mathtt{undef} \wedge x_1 = f_1(x_0) \wedge \cdots \wedge x_n = f_n(x_0) \tag{2}$$

Hence, the representation of negated atoms is the one that directly follows from negating the formula in (2):

$$\neg R_S(x_0, \ldots, x_n) \quad \text{iff} \quad x_0 = \mathtt{undef} \vee x_1 \neq f_1(x_0) \vee \cdots \vee x_n \neq f_n(x_0)$$

This relational interpretation of DB schemas exactly reconstructs the requirements posed by [32,50] 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 the one in Axiom (1). In the following subsection, 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 [17, 32]. 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 [32] towards the main decidability results.

3.2 Formal Properties of DB Schemas

The theory T from Definition 3.1 must satisfy a few crucial requirements for our approach to work. In this section, we define such requirements and show that they are matched in the cases we are interested in. The following proposition is motivated by the fact that in most cases the kind of axioms that we need for our DB theories T are just *one-variable universal axioms* (like Axioms (1)).

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.[6] The finite model property implies decidability of the constraint satisfiability problem for T if T is recursively axiomatized.

Proposition 3.6. *T has the finite model property and has a model completion in case it is axiomatized by universal one-variable formulae and Σ is acyclic.*

The proof of the above result in [24] supplies an algorithm for quantifier elimination in the model completion which is far from optimal in concrete cases. Moreover, acyclicity is not needed in general for Proposition 3.6 to hold: for instance, when $T := \emptyset$ or when T contains only Axioms (1), the proposition holds without acyclicity hypothesis. Such improvements are explained in [23], where a better quantifier elimination algorithm, based on Knuth-Bendix completion is supplied. Proposition 3.6 nevertheless motivates the following assumption:

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

This assumption is matched, for instance, in the following three cases: *(i)* when T is empty; *(ii)* when T is axiomatized by Axioms (1); *(iii)* when Σ is acyclic and T is axiomatized by finitely many universal one-variable formulae (such as Axioms (1)).

Hence, the artifact-centric model in the style of [32, 50] that we intend to capture *matches* Assumption 1.

[6] This directly implies that ϕ is satisfiable also in a DB instance that interprets value sorts into finite sets.

4 Relational Artifact Systems

We are now in the position to define our formal model of *Relational Artifact Systems* (RASs), and to study parameterized safety problems over RASs. Since RASs are array-based systems, we start by recalling the intuition behind them.

In general terms, an array-based system is described using a multi-sorted theory that contains two types of sorts, one accounting for the indexes of arrays, and the other for the elements stored therein. Since the content of an array changes over time, it is referred to by a second-order function variable, whose interpretation in a state is that of a total function mapping indexes to elements (so that applying the function to an index denotes the classical *read* operation for arrays). The definition of an array-based system with array state variable a always requires a formula $I(a)$ describing the *initial configuration* of the array a, and a formula $\tau(a, a')$ describing a *transition* that transforms the content of the array from a to a'. In such a setting, verifying whether the system can reach unsafe configurations described by a formula $K(a)$ amounts to checking whether the formula $I(a_0) \wedge \tau(a_0, a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge K(a_n)$ is satisfiable for some n. Next, we make these ideas formally precise by grounding array-based systems in the artifact-centric setting.

Following the tradition of artifact-centric systems [17,28,31,32], a RAS consists of a read-only DB, a read-write working memory for artifacts, and a finite set of actions (also called services) that inspect the relational database and the working memory, and determine the new configuration of the working memory. In a RAS, the working memory consists of *individual* and *higher order* variables. These variables (usually called *arrays*) are supposed to model evolving relations, so-called *artifact relations* in [32,50]. The idea is to treat artifact relations in a uniform way as we did for 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[7]. 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*. In a RAS, artifacts and basic sorts correspond, respectively, to the index and the elements sorts mentioned in the literature on array-based systems. 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 $\underline{x}$ of individual variables and a finite set $\underline{a}$ of unary function variables: *the latter must have an artifact sort as source sort and a basic sort as target sort.* Variables in $\underline{x}$ are called *artifact variables*, and variables in $\underline{a}$ *artifact components*. Given a DB

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

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 $x_i^\alpha \in S_i^{\mathcal{M}}$ and to every artifact component $a_j : E_j \longrightarrow U_j$ (with $a_j \in \underline{a}$) a set-theoretic function $a_j^\alpha : E_j^{\mathcal{M}} \longrightarrow U_j^{\mathcal{M}}$. In a RAS, artifact components and artifact variables correspond, respectively, to *arrays* and *constant arrays* (i.e., arrays with all equal elements) mentioned in the literature on array-based systems.

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 $\{\langle e, a_{i_1}^\alpha(e), \ldots, a_{i_n}^\alpha(e)\rangle \mid e \in E^{\mathcal{M}}\}$. Thus each element of E is formed by an "entry" $e \in E^{\mathcal{M}}$ (uniquely identifying the tuple) and by "data" $\underline{a}_i^\alpha(e)$ taken from the read-only database $\mathcal{M}$. When the system evolves, the set $E^{\mathcal{M}}$ of entries remains fixed, whereas the components $\underline{a}_i^\alpha(e)$ may change: typically, we initially have $\underline{a}_i^\alpha(e) = \mathtt{undef}$, 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)[8].

In order to introduce verification problems in the symbolic setting of array-based systems, one first has to specify which formulae are used to represent sets of states, the system initializations, and system evolution. 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 implicit).

- An *initial formula* is a formula $\iota(\underline{x})$ of the form[9]

$$(\textstyle\bigwedge_{i=1}^n x_i = c_i) \;\wedge\; (\bigwedge_{j=1}^m a_j = \lambda y.d_j),$$

  where c_i, d_j are constants from Σ (typically, c_i and d_j are `undef`).
- A *state formula* has the form

$$\exists \underline{e}\, \phi(\underline{e}, \underline{x}, \underline{a}),$$

  where ϕ is quantifier-free and the $\underline{e}$ are individual variables of artifact sorts.
- A *transition formula* $\hat{\tau}$ has the form

$$\exists \underline{e}\, (\gamma(\underline{e}, \underline{x}, \underline{a}) \;\wedge\; \textstyle\bigwedge_i x_i' = F_i(\underline{e}, \underline{x}, \underline{a}) \;\wedge\; \bigwedge_j a_j' = \lambda y.G_j(y, \underline{e}, \underline{x}, \underline{a})) \qquad (3)$$

[8] 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 $a[v]$ (standard notation for arrays), instead of $a(v)$.

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

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.

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)* bulk removal/update of *all* the tuples satisfying a certain condition from an artifact relation. All the above operations can also be constrained. Our framework is more expressive than, e.g., the one in [50], as shown in [24].

Definition 4.1. *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 4.2. We present here 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 to accept or reject. The example is inspired by the job hiring process presented in [58] to show the intrinsic difficulties of capturing real-life processes with many-to-many interacting business entities using conventional process modeling notations (e.g., BPMN). An extended version of this example is presented in [24].

As for the read-only DB, $\mathcal{S}_{hr}$ works over the DB schema of Example 3.3, 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 readability, we use as syntactic sugar the usual predicates $<$, $>$, and $=$ to compare variables of type Score.

As for the working memory, $\mathcal{S}_{hr}$ consists of two artifacts. The first single-instance *job hiring* artifact employs a dedicated *pState* variable to capture main phases that the running process goes through: initially, hiring is disabled (*pState* $=$ `undef`), and, if there is at least one registered user in the HR DB, *pState* becomes enabled. The second multi-instance artifact accounts for the evolution of *user applications*. To model applications, we take the DB signature Σ_{hr} of the read-only HR DB, 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 storing different job applications; *(ii)* additional individual variables as 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 the final result (indicating whether the application is accepted or not). These information slots are encapsulated into dedicated artifact components, i.e., function variables with domain $\mathsf{applndex}$ that collectively realize the application artifact relation:

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

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 means that it is updated identically; otherwise, the relevant update function will specify how it is updated.[10] The insertion of an application into the system can be executed when the hiring process is enabled, 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 (user ID, job category ID, and employee ID) into dedicated artifact variables (*uId*, *jId*, *eId*, respectively) and evolves *aState* into state `received`.

The second step transfers the application data into the application artifact relation (using its corresponding function variables), and resets 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` ("not eligible"), and an `undef` final result:

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

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 [24] for an example).

[10] Non-deterministic updates can be formalized using existentially quantified variables in the transition.

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

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

Finally, application results are computed when the process moves to state `notified`. This is handled by the *bulk* transition:

$$pState = \texttt{enabled} \wedge pState' = \texttt{notified} \wedge \\ appResult' = \lambda j. \,(\mathsf{if}\ appScore[j] > \mathsf{80}\ \mathsf{then}\ \texttt{winner}\ \mathsf{else}\ \texttt{loser})$$

which declares applications with a score above 80 as winning, and the others as losing. $\triangleleft$

5 Parameterized Safety via Backward Reachability

A *safety* formula for $\mathcal{S}$ is a state formula $\upsilon(\underline{x})$ describing undesired states of $\mathcal{S}$. As usual in array-based systems, 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 $\langle \Sigma_{ext}, T\rangle$, 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

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

is true in $\mathcal{M}$ (here $\underline{x}^i$, $\underline{a}^i$ are renamed copies of $\underline{x}$, $\underline{a}$). The *safety problem* for $\mathcal{S}$ is the following: *given a safety formula* υ *decide whether* $\mathcal{S}$ *is safe with respect to* υ.

Example 5.1. The following property expresses the undesired situation that, in the RAS from Example 4.2, once the evaluation is notified there is an applicant with unknown result:

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

The job hiring RAS $\mathcal{S}_{hr}$ turns out to be safe with respect to this property (cf. Sect. 7). $\triangleleft$

We shall introduce an algorithm that semi-decides safety problems for $\mathcal{S}$, and in the next section we shall examine some interesting cases where the algorithm terminates and gives a decision procedure. Before introducing the algorithm, we need some technical results specifying how far we can extend the T^*-quantifier elimination procedure and the T-satisfiability procedure for Σ-constraints to a larger class of quantified formulae in the enriched signature of our artifact settings.

An integral part of the algorithm is to compute *symbolic* preimages. For that purpose, we define for any $\phi_1(\underline{z}, \underline{a}, \underline{z}', \underline{a}')$ and $\phi_2(\underline{z}, \underline{a})$, $Pre(\phi_1, \phi_2)$ as the formula $\exists \underline{z}' \exists \underline{a}' (\phi_1(\underline{z}, \underline{a}, \underline{z}', \underline{a}') \wedge \phi_2(\underline{z}', \underline{a}'))$. The *preimage* of the set of states described by a state formula $\phi(\underline{x}, \underline{a})$ is the set of states described by $Pre(\tau, \phi)$ (notice that, when $\tau = \bigvee \hat{\tau}$, we have $Pre(\tau, \phi) = \bigvee Pre(\hat{\tau}, \phi)$).

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. The next two lemmas are proved via syntactic manipulations:

Algorithm 1. Schema of the backward reachability algorithm

```
Function BReach($\upsilon$)
1   $\phi \longleftarrow \upsilon$; $B \longleftarrow \bot$;
2   while $\phi \wedge \neg B$ is $T$-satisfiable do
3       if $\iota \wedge \phi$ is $T$-satisfiable then
            return unsafe
4       $B \longleftarrow \phi \vee B$;
5       $\phi \longleftarrow Pre(\tau, \phi)$;
6       $\phi \longleftarrow$ QE$(T^*, \phi)$;
    return (safe, $B$);
```

Lemma 5.2. *The preimage of a state formula is logically equivalent to an extended state formula.*

Lemma 5.3. *For every extended state formula ϕ there is a state formula* QE(T^*, ϕ) *equivalent to ϕ in all Σ_{ext}-models of T^*.*

We underline that Lemmas 5.2 and 5.3 both give an explicit effective procedure for computing equivalent (extended) state formulae: such effective procedures will be an essential part of our backward reachability algorithm. Notice that Lemma 5.3 relies on quantifier elimination in T^*, in fact it is meant to eliminate existentially quantified variables *ranging over basic sorts*. Existentially quantified variables over artifact sorts, on the contrary, cannot be eliminated as they occur as arguments of artifact components.

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

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

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 quantified variables in $\exists\forall$-formulae are all of artifact sorts (the lemma is proved by syntactic manipulations followed by suitable instantiations):

Lemma 5.4. *The satisfiability of an $\exists\forall$-formula in a Σ_{ext}-model of T is decidable. Moreover, an $\exists\forall$-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^*.*

Algorithm 1 describes the *backward reachability algorithm* (or, *backward search*) for handling the safety problem for $\mathcal{S}$. It computes iterated preimages of υ and applies to them the procedures from Lemmas 5.2 and 5.3, until a fixpoint is reached or until a set intersecting the initial states (i.e., satisfying ι) is found. The satisfiability tests from Lines 2 and 3 can be effectively discharged by

Lemma 5.4 (in fact, the procedure of Lemma 5.4 reduces them to T-constraint satisfiability problems).

To sum up, we obtain the following theorem (to understand the statement of the theorem, notice that by *partial correctness* we mean that, when the algorithm terminates, it gives a correct answer, and by *effectiveness* we mean that all subprocedures in the algorithm can be effectively executed):

Theorem 5.5. *Backward search (cf. Algorithm 1) is effective and partially correct for solving safety problems for RASs.*

Theorem 5.5 shows that backward search is 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). 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 5.4); *(ii)* it ensures termination in significant cases, namely those where *(strongly) local formulae*, introduced in the next section, are involved.

6 Termination Results for RASs

We now present three termination results, two relating RASs to previous fundamental results, and one genuinely novel.

Termination for "Simple" Artifact Systems. An interesting class of RASs is the one where the working memory consists *only* of artifact variables (without artifact relations). We call systems of this type SASs (*Simple Artifact Systems*). For SASs, the following termination result holds.

Theorem 6.1. *Let $\langle \Sigma, T\rangle$ be a DB schema with Σ acyclic. Then, for every SAS $\mathcal{S} = \langle \Sigma, T, \underline{x}, \iota, \tau\rangle$, backward search terminates and decides safety problems for $\mathcal{S}$ in* Pspace *in the combined size of $\underline{x}$, ι, and τ.*

It is worth noticing that the decidability part of Theorem 6.1 can be easily extended to *locally finite theories* T (thus, in particular to *arbitrary relational signatures*) whenever T has the amalgamation property and is closed under substructures. Thanks to these observations, Theorem 6.1 is reminiscent of an analogous result in [17], i.e., Theorem 5, the crucial hypotheses of which are exactly amalgamability and closure under substructures, although the setting in that paper is different (there, key dependencies are not discussed, but there is no limitation to elementarily definable classes of structures). Notice also that a distinctive feature of our framework is that it remains well-behaved even in the presence of key dependencies (a naive representation of primary key dependencies with partially functional relations would cause amalgamability to fail). Another important point is that we perform verification in a purely symbolic way, using decision procedures provided by SMT-solvers.

Termination with Local Updates. Consider an *acyclic* signature Σ *not containing relation symbols*, a DB 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

$$\exists e_1 \cdots \exists e_k \, (\delta(e_1, \ldots, e_k) \wedge \bigwedge_{i=1}^{k} \phi_i(e_i, \underline{x}, \underline{a})), \tag{4}$$

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

$$\exists e_1 \cdots \exists e_k \, (\delta(e_1, \ldots, e_k) \wedge \psi(\underline{x}) \wedge \bigwedge_{i=1}^{k} \phi_i(e_i, \underline{a})). \tag{5}$$

In (4) and (5), δ is a conjunction of variable equalities and inequalities, ϕ_i, ψ are quantifier-free, and $e_1, \ldots, e_k$ are individual variables ranging over artifact sorts. The key limitation of local state formulae is that they cannot compare entries from different tuples of artifact relations: each ϕ_i in (4) and (5) 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 [24].

Theorem 6.2. *If Σ is acyclic and does not contain relation symbols, 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 $\Sigma_{ext} \cup \{\underline{a}, \underline{x}\}$, i.e., Σ_{ext} expanded with function symbols $\underline{a}$ and constants $\underline{x}$ ($\underline{a}$ and $\underline{x}$ are treated as symbols of $\tilde{\Sigma}$, but not as variables anymore). We call a $\tilde{\Sigma}$-structure *cyclic*[11] 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 isomorphic to $\mathcal{C}_1, \ldots, \mathcal{C}_N$, 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 [13]. $\dashv$

Note that Theorem 6.2 can be used to reconstruct the decidability results of [50] concerning safety problems. Specifically, one needs to show that transitions in [50] are strongly local which, in turn, can be shown using quantifier elimination (see [24] for more details). Interestingly, Theorem 6.2 can be applied to more cases not covered in [50]. For example, one can provide transitions enforcing *updates over unboundedly many tuples* (bulk updates) that are strongly local. 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 [30,35], which can be encoded using local formulae over the trivial one-sorted signature containing just one basic sort, finitely many constants, and one artifact sort with one artifact component. These problems can

[11] This is unrelated to cyclicity of Σ defined in Sect. 3, and comes from universal algebra terminology.

be decided with a non-primitive recursive lower bound [57] (whereas the problems in [50] have an ExpSpace upper bound). Recalling that [50] handles verification of LTL-FO, thus going beyond safety problems, this shows that the two settings are incomparable. Notice that Theorem 6.2 implies also the decidability of the safety problem for SASs, in case of acyclic Σ.

Termination for Tree-like Signatures. Σ is *tree-like* if it is acyclic, does not contain relation symbols, and all non-leaf nodes 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 6.3 *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 is to show, using Kruskal's Tree Theorem [47], that the finitely generated $\tilde{\Sigma}$-structures are a well-quasi-order w.r.t. the embeddability partial order. $\dashv$

While tree-like RAS restrict artifact relations to be unary, their transitions are not subject to any locality restriction. This allows for expressing rich forms of updates, including general bulk updates (which allow us to capture non-primitive recursive verification problems) and transitions comparing at once different tuples in artifact relations. Notice that tree-like RASs are incomparable with the "tree" classes of [17], since the former use artifact relations, whereas the latter only individual variables. In [24] we show the power of such advanced features in a flight management process example.

7 First Experiments

We implemented a prototype of the backward reachability algorithm for RASs on top of the mcmt model checker for array-based systems. Starting from its first version [42], mcmt was successfully applied to a variety of settings: cache coherence and mutual exclusions protocols [41], timed [25] and fault-tolerant [5,6] distributed systems, and imperative programs [7,8]. Interesting case studies concerned waiting time bounds synthesis in parameterized timed networks [19] and internet protocols [18]. Further related tools include safari [3], asasp [2], and Cubicle [27]. The latter relies on a parallel architecture with further powerful extensions. The work principle of mcmt 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 [34]). 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.

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

Exp.	#AC	#AV	#T	Prop.	Res.	Time (sec)
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

We produced a benchmark consisting of eight realistic business process examples and ran it in MCMT (detailed explanations and results are given in [24]). The examples are partially made by hand and partially obtained from those supplied in [50]. A thorough comparison with VERIFAS [50] is matter of future work, and is non-trivial for a variety of reasons. In particular, as already mentioned in Sect. 6, the two systems tackle incomparable verification problems: on the one hand, we deal with safety problems, whereas VERIFAS handles more general LTL-FO properties; on the other hand, we tackle features not available in VERIFAS, like bulk updates and comparisons between artifact tuples. Moreover, the two verifiers implement completely different state space construction strategies: MCMT is based on backward reachability and makes use of declarative techniques that rely on decision procedures, while VERIFAS employs forward search via VASS encoding.

The benchmark set is available as part of the last distribution 2.8 of MCMT.[12] Table 1 shows the very encouraging results (the first row tackles Example 5.1). While a systematic evaluation is out of scope of this paper, MCMT effectively solves the benchmarks with a comparable performance shown in other well-studied areas, with verification times below 1s in most cases.

8 Conclusions

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 exploited the model-theoretic machinery of model completion to overcome the main technical difficulty arising from this approach, i.e., showing how to reconstruct quantifier elimination in the rich setting of artifact systems. On top of this framework, we have identified three classes of systems

[12] http://users.mat.unimi.it/users/ghilardi/mcmt/, subdirectory `/examples/dbdriven` of the distribution. The user manual contains a new section (pages 36–39) on how to encode RASs in MCMT specifications.

for which safety is decidable, which impose different combinations of restrictions on the form of actions and the shape of DB constraints. The presented techniques have been implemented on top of the well-established MCMT model checker, making our approach fully operational.

We consider the present work as the starting point for a full line of research dedicated to SMT-based techniques for the effective verification of data-aware processes, addressing richer forms of verification beyond safety (such as liveness, fairness, or full LTL-FO) and richer classes of artifact systems, (e.g., with concrete data types and arithmetics), while identifying novel decidable classes (e.g., by restricting the structure of the DB and of transition and state formulae). Concerning implementation, we plan to further develop our tool to incorporate in it the plethora of optimizations and sophisticated search strategies available in infinite-state SMT-based model checking. Finally, we plan to tackle more conventional process modeling notations, concerning in particular data-aware extensions of the de-facto standard BPMN[13].

References

1. Abdulla, P.A., Aiswarya, C., Atig, M.F., Montali, M., Rezine, O.: Recency-bounded verification of dynamic database-driven systems. In: Proceedings of the PODS, pp. 195–210 (2016)
2. Alberti, F., Armando, A., Ranise, S.: ASASP: automated symbolic analysis of security policies. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 26–33. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_4
3. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-based abstraction for arrays with interpolants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_49
4. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: An extension of lazy abstraction with interpolation for programs with arrays. Formal Methods Syst. Des. **45**(1), 63–109 (2014)
5. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Brief announcement: automated support for the design and validation of fault tolerant parameterized systems - a case study. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 392–394. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15763-9_36
6. Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., Rossi, G.P.: Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. J. Satisfiability Boolean Model. Comput. 8(1/2), 29–61 (2012)
7. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: an acceleration-based verification framework for array programs. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 18–23. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_2
8. Alberti, F., Ghilardi, S., Sharygina, N.: A framework for the verification of parameterized infinite-state systems. Fundam. Inf. **150**(1), 1–24 (2017)

[13] http://www.bpmn.org/.

9. Baader, F., Ghilardi, S.: Connecting many-sorted structures and theories through adjoint functions. In: Gramlich, B. (ed.) FroCoS 2005. LNCS (LNAI), vol. 3717, pp. 31–47. Springer, Heidelberg (2005). https://doi.org/10.1007/11559306_2
10. Baader, F., Ghilardi, S.: Connecting many-sorted theories. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 278–294. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_21
11. Baader, F., Ghilardi, S.: Connecting many-sorted theories. J. Symbolic Logic **72**(2), 535–583 (2007)
12. Baader, F., Ghilardi, S., Tinelli, C.: A new combination procedure for the word problem that generalizes fusion decidability results in modal logics. Inf. Comput. **204**(10), 1413–1452 (2006)
13. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
14. Baader, F., Tinelli, C.: Deciding the word problem in the union of equational theories. Inf. Comput. **178**(2), 346–390 (2002)
15. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. In: Proceedings of the PODS, pp. 163–174 (2013)
16. Belardinelli, F., Lomuscio, A., Patrizi, F.: An abstraction technique for the verification of artifact-centric systems. In: Proceedings of the KR (2012)
17. Bojańczyk, M., Segoufin, L., Toruńczyk, S.: Verification of database-driven systems via amalgamation. In: Proceedings of the PODS, pp. 63–74 (2013)
18. Bruschi, D., Di Pasquale, A., Ghilardi, S., Lanzi, A., Pagani, E.: Formal verification of ARP (address resolution protocol) through SMT-based model checking - a case study. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 391–406. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_26
19. Bruttomesso, R., Carioni, A., Ghilardi, S., Ranise, S.: Automated analysis of parametric timing-based mutual exclusion algorithms. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 279–294. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_28
20. Calvanese, D. ., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: a database theory perspective. In: Proceedings of the PODS, pp. 1–12 (2013)
21. Calvanese, D., De Giacomo, G., Montali, M., Patrizi, F.: First-order mu-calculus over generic transition systems and applications to the situation calculus. Inf. Comput. **259**, 328–347 (2017)
22. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Model completeness for the verification of data-aware processes. Manuscript submitted for publication (2018)
23. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Quantifier elimination for database driven verification. Technical report arXiv:1806.09686, arXiv.org (2018)
24. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Verification of data-aware processes via array-based systems (extended version). Technical report arXiv:1806.11459, arXiv.org (2018)
25. Carioni, A., Ghilardi, S., Ranise, S.: MCMT in the land of parametrized timed automata. In: Proceedings of the VERIFY. EPiC Series in Computing, vol. 3, pp. 47–64 (2010)
26. Chang, C.-C., Keisler, J.H.: Model Theory. North-Holland Publishing Co. (1990)

27. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_55
28. Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic. ACM TODS **37**(3), 22 (2012)
29. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. Inf. Syst. **38**(4), 561–584 (2013)
30. Delzanno, G., Podelski, A., Esparza, J.: Constraint-based analysis of broadcast protocols. In: Flum, J., Rodriguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 50–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48168-0_5
31. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proceedings of the ICDT, pp. 252–267. ACM (2009)
32. Deutsch, A., Li, Y., Vianu, V.: Verification of hierarchical artifact systems. In: Proceedings of the PODS, pp. 179–194 (2016)
33. Dumas, M.: On the convergence of data and process engineering. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 19–26. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23737-9_2
34. Dutertre, B., De Moura, L.: The YICES SMT solver. Technical report, SRI International (2006)
35. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Proceedings of the LICS, pp. 352–359. IEEE Computer Society (1999)
36. Fiorentini, C., Ghilardi, S.: Combining word problems through rewriting in categories with products. TCS **294**(1–2), 103–149 (2003)
37. Ghilardi, S.: Model theoretic methods in combined constraint satisfiability. JAR **33**(3–4), 221–249 (2004)
38. Ghilardi, S., Gianola, A.: Interpolation, amalgamation and combination (the non-disjoint signatures case). In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 316–332. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_18
39. Ghilardi, S., Gianola, A.: Modularity results for interpolation, amalgamation and superamalgamation. Ann. Pure Appl. Logic **169**(8), 731–754 (2018)
40. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 67–82. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_6
41. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. Log. Methods Comput. Sci. **6**(4) (2010)
42. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
43. Ghilardi, S., van Gool, S.J.: Monadic second order logic as the model companion of temporal logic. In: Proceedings of the LICS, pp. 417–426. ACM (2016)
44. Ghilardi, S., van Gool, S.J.: A model-theoretic characterization of monadic second order logic on infinite words. J. Symbolic Logic **82**(1), 62–76 (2017)
45. Gulwani, S., Musuvathi, M.: Cover algorithms and their combination. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 193–207. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_16

46. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88873-4_17
47. Kruskal, J.B.: Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. Trans. Amer. Math. Soc. **95**, 210–225 (1960)
48. Künzle, V., Weber, B., Reichert, M.: Object-aware business processes: fundamental requirements and their support in existing approaches. Int. J. Inf. Syst. Model. Des. **2**(2), 19–46 (2011)
49. Kutz, O., Lutz, C., Wolter, F., Zakharyaschev, M.: E-connections of abstract description systems. AIJ **156**(1), 1–73 (2004)
50. Li, Y., Deutsch, A., Vianu, V.: VERIFAS: a practical verifier for artifact systems. PVLDB **11**(3), 283–296 (2017)
51. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM TOPLAS **1**(2), 245–257 (1979)
52. Pigozzi, D.: The join of equational theories. Colloq. Math. **30**, 15–25 (1974)
53. Reichert, M.: Process and data: two sides of the same coin? In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7565, pp. 2–19. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_2
54. Richardson, C.: Warning: don't assume your business processes use master data. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 11–12. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15618-2_3
55. Robinson, A.: On the Metamathematics of Algebra. North-Holland (1951)
56. Robinson, A.: Introduction to model theory and to the metamathematics of algebra. In: Studies in Logic and the Foundations of Mathematics. North-Holland (1963)
57. Schmitz, S., Schnoebelen, P.: The power of well-structured systems. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 5–24. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_2
58. Silver, B.: BPMN Method and Style. 2nd edn. Cody-Cassidy (2011)
59. Sofronie-Stokkermans, V.: On interpolation and symbol elimination in theory extensions. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 273–289. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_19
60. Sofronie-Stokkermans, V.: On interpolation and symbol elimination in theory extensions. Log. Methods Comput. Sci. **14**(3) (2018)
61. Tinelli, C., Harandi, M.: A new correctness proof of the nelson-oppen combination procedure. In: Baader, F., Schulz, K.U. (eds.) Frontiers of Combining Systems. ALS, vol. 3, pp. 103–119. Springer, Dordrecht (1996). https://doi.org/10.1007/978-94-009-0349-4_5
62. Vianu, V.: Automatic verification of database-driven systems: a new frontier. In: Proceedings of the ICDT, pp. 1–13. ACM (2009)
63. Wolter, f.: Fusions of modal logics revisited. In: Advances in Modal Logic. CSLI Lecture Notes, vol. 1, pp. 361–379 (1996)