UNIVERSITY OF CALIFORNIA SAN DIEGO

Verification of Hierarchical Data-Driven Workflows

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Yuliang Li

2018

The Dissertation of Yuliang Li is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California San Diego

2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to convey my heartfelt gratitude to my PhD advisors Professor Alin Deutsch and Professor Victor Vianu. I feel very fortunate to have them as my advisors. Over the past 6 years, they have been very good friends and mentors, guiding me in overcoming obstacles in both life and research. My days as their student were wonderful and enlightening.

Besides my advisors, my sincere thanks also go to the rest of my thesis committee: Professor Samuel Buss, Professor Sorin Lerner, and Professor Jianwen Su, for their constructive feedback/comments that helped complete my thesis and dissertation.

I also would like to thank Hongrae Lee who hosted me in Google Research during the summer of 2013, Christian König who was the mentor of my internship in Microsoft Research during the summer of 2014, and Richard Hull who hosted me in IBM Thomas J. Watson Research Center during the summer of 2017. These internships and great mentors brought me valuable experiences with the industrial research world and my summers would not be so great without you.

I thank my fellow labmates for the insightful discussions, all the fun, and coffee we have had. In particular, I would like to thank Yanqin Jin, Jing Li, Chunbin Lin and Hung-Wei Tseng. Life is nothing without friends.

Finally, I would not be able to achieve anything without the help and support from my beloved wife Shuang Song and my family.

Chapter 3 of this thesis contains material from "Verification of Hierarchical Artifact Systems" by Alin Deutsch, Yuliang Li and Victor Vianu, which appears in Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2016). The dissertation author was the primary investigator of this paper.

Chapter 4 contains material from "SpinArt: A Spin-based Verifier for Artifact Systems" by Yuliang Li, Alin Deutsch and Victor Vianu. This paper is in ArXiv. The dissertation author was the primary investigator of this paper.

Chapter 5 contains material from “VERIFAS: A Practical Verifier for Artifact Systems” by Yuliang Li, Alin Deutsch and Victor Vianu in Proceedings of VLDB Endowment 11, 3 (November 2017). The dissertation author was the primary investigator of this paper.

VITA

2012	Bachelor of Engineering, the Hong Kong University of Science and Technology, Hong Kong
2015	Master of Science, University of California San Diego
2018	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Automatic verification of database-centric systems. ACM SIGLOG News 5, 2 (April 2018), 37-56

VERIFAS: a practical verifier for artifact systems. Proc. VLDB Endow. 11, 3 (November 2017), 283-296

Practical Verification of Hierarchical Artifact Systems. Proceedings of the VLDB 2017 PhD Workshop co-located with VLDB 2017

SpinArt: A Spin-based Verifier for Artifact Systems. arXiv preprint, 2017

Verification of Hierarchical Artifact Systems. In Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '16). ACM, New York, NY, USA, 179-194

Mining order-preserving submatrices from probabilistic matrices. ACM Trans. Database Syst. 39, 1, Article 6 (January 2014), 43 pages

Mining Bucket Order-Preserving SubMatrices in Gene Expression Data. IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 24 no. 12, Dec. 2012

ABSTRACT OF THE DISSERTATION

Verification of Hierarchical Data-Driven Workflows

by

Yuliang Li

Doctor of Philosophy in Computer Science

University of California San Diego, 2018

Professor Alin Deutsch, Co-Chair
Professor Victor Vianu, Co-Chair

Data-driven workflows, of which IBM's Business Artifacts are a prime exponent, have been successfully deployed in practice, adopted in industrial standards, and have spawned a rich body of research in academia, focused primarily on static analysis. This thesis represents a significant advance on the problem of verifying data-driven workflows in two major aspects.

First, this thesis introduces Hierarchical Artifact Systems (HAS), a much richer and more realistic model than previously considered, incorporating core elements of IBM's successful Guard-Stage-Milestone model. In particular, the HAS model features task hierarchy, concurrency, and richer artifact data. It also allows database key and foreign key dependencies, as well as arithmetic constraints. The results show decidability of verification and establish its complexity under a set of reasonable restrictions, making use of novel techniques including a hierarchy of Vector Addition Systems and a variant of quantifier elimination tailored to this context.

Second, this thesis bridges the gap between the theory and practice of data-driven workflow verification with two successful implementations, SpinArt and VERIFAS. SpinArt is a practical verifier based on the classical model-checking tool Spin, and can verify a core subset of the HAS model. The implementation includes nontrivial optimizations and achieves good performance on real-world business process examples. VERIFAS further bridges the gap with a specialized implementation built from scratch. It verifies within seconds linear-time temporal properties over both real-world and synthetic workflows of complexity in the range recommended by software engineering practice. Compared to SpinArt, VERIFAS not only supports a model with richer data manipulations but also outperforms it by over an order of magnitude. VERIFAS's good performance is due to a novel symbolic representation approach and a family of specialized optimizations. To the best of our knowledge, these are the first practically significant implementations of artifact verifiers, that provide full support for unbounded data.

Chapter 1

Introduction

The past decade has witnessed the evolution of workflow specification frameworks from the traditional process-centric approach towards data-awareness. Process-centric formalisms focus on control flow while under-specifying the underlying data and its manipulations by the process tasks, often abstracting them away completely. In contrast, data-aware formalisms treat data as first-class citizens. A notable exponent of this class is IBM's *business artifact model* pioneered in [103], successfully deployed in practice [16, 14, 33, 40, 130] and adopted in industrial standards. Business artifacts have also spawned a rich body of research in academia, dealing with issues ranging from formal semantics to static analysis.

In a nutshell, business artifacts (or simply "artifacts") model key business-relevant entities, which are updated by a set of services that implement business process tasks, specified declaratively by pre-and-post conditions. A collection of artifacts and services is called an *artifact system*. IBM has developed several variants of artifacts, of which the most recent is Guard-Stage-Milestone (GSM) [37, 76]. The GSM approach provides rich structuring mechanisms for services, including parallelism, concurrency and hierarchy, and has been incorporated in the OMG standard for Case Management Model and Notation (CMMN) [17, 94].

Artifact systems deployed in industrial settings typically specify very complex workflows that are prone to costly bugs, whence the need for verification of critical properties. Over the past few years, an active line of research on the verification of artifact systems has emerged.

Rather than relying on general-purpose software verification tools suffering from well-known limitations, the aim of this thesis, described below, is to identify practically relevant classes of artifact systems and properties for which *fully automatic* verification is possible. This is an ambitious goal, since artifacts are infinite-state systems due to the presence of unbounded data. Approaches to this problem rely critically on the declarative nature of service specifications, bringing into play a novel marriage of database and computer-aided verification techniques.

In previous work [44, 36], the verification problem was studied for a bare-bones variant of artifact systems, without hierarchy or concurrency, in which each artifact consists of a flat tuple of evolving values and the services are specified by simple pre-and-post conditions on the artifact and database. More precisely, the problem considered was to statically check whether all runs of an artifact system satisfy desirable properties expressed in LTL-FO, an extension of linear-time temporal logic where propositions are interpreted as $\exists$FO sentences on the database and current artifact tuple. In order to deal with the resulting infinite-state system, in [44], a symbolic approach was developed to allow a reduction to finite-state model checking and yielding a PSPACE verification algorithm for the simplest variant of the model (no database dependencies and uninterpreted data domain). In [36] the approach was extended to allow for database dependencies and numeric data testable by arithmetic constraints. Unfortunately, decidability was obtained subject to a rather complex semantic restriction on the artifact system and property (feedback freedom), and the verification algorithm has non-elementary complexity.

The present thesis represents a significant advance on the artifact verification problem on several fronts. It considers a much richer and more realistic model, called *Hierarchical Artifact System* (HAS), abstracting core elements of the GSM model. In particular, the model features task hierarchy, concurrency, and richer artifact data (including updatable artifact relations). In brief, a HAS consists of a database and a hierarchy (rooted tree) of *tasks*. Each task has associated to it local evolving data consisting of a tuple of artifact variables and an updatable artifact relation. It also has an associated set of *services*. Each application of a service is guarded by a pre-condition on the database and local data and causes an update of the local data, specified by a post condition (constraining the next artifact tuple) and an insertion or retrieval of a tuple from the artifact relation. In addition, a task may invoke a child task with a tuple of parameters, and receive back a result if the child task completes. A run of the artifact system is obtained by any valid interleaving of concurrently running task services. For verification, properties of HAS are expressed in a novel *hierarchical* temporal logic, HLTL-FO, that is well-suited to the model.

The main results establish the complexity of checking HLTL-FO properties for various classes of HAS, highlighting the impact of various features on verification. The results require qualitatively novel techniques, because the reduction to finite-state model checking used in previous work is no longer possible. Instead, the richer model requires the use of a hierarchy of Vector Addition Systems with States (VASS) [18]. The arithmetic constraints are handled using quantifier elimination techniques, adapted to the HAS and HLTL-FO setting. The discussion of the HAS model, HLTL-FO and the theoretical results are presented in Chapter 3.

Chapter 4 and Chapter 5 of the thesis study the practical implementation of data-driven workflows verifiers. The first presented implementation is *SpinArt* in Chapter 4, a fully automatic verifier for artifact systems. The goal in this implementation is to explore the feasibility of using existing off-the-shelf tools to build such a practical verifier. The implementation focuses specifically on Spin [74], the main model checker used in the verification community and the natural candidate for a verifier implementation. SpinArt verifies a core fragment of HAS called the *Tuple Artifact Systems* (TAS). The model is expressive enough to allow a database of arbitrary size, which is not directly supported by Spin or other state-of-the-art model checkers. SpinArt addresses this challenge by exploiting the symbolic verification techniques establishing the decidability results for HAS. Together with an array of optimization techniques, SpinArt renders verification tractable. To the best of our knowledge, SpinArt is the first implementation of an artifact verifier that preserves decidability under unbounded data while being based on off-the-shelf model checking technology.

This first attempt of implementation shed light on the capabilities and limitations of off-the-shelf verification tools in the context of data-driven workflows. For example, SpinArt cannot handle some of the most useful features supporting the unbounded data, such as the updatable artifact relations (i.e. updatable sets of tuples). Moreover, even after deploying a set of non-trivial optimizations, the performance of SpinArt is still unsatisfactory because of intrinsic limitations of Spin (shared by similar tools) and missing problem-specific optimization opportunities. This indicates the restricted applicability of existing tools for HAS verification and suggests the need for tailored approaches.

In view of these findings, Chapter 5 of the thesis presents VERIFAS, a workflow verifier implementation built from scratch. The main contributions are the following. VERIFAS verifies a novel variant of HAS called HAS*, which strikes a more practically relevant trade-off between expressivity and verification complexity. This is demonstrated by its ability to specify a realistic set of business processes. We adapt to HAS* the theory developed for HAS, laying the groundwork for the implementation. To achieve good performance, VERIFAS makes use of a set of crucial optimizations with dramatic impact, including concise symbolic representations, aggressive pruning in search algorithms, and the use of highly efficient data structures.

The performance of SpinArt and VERIFAS is evaluated and compared on both real-world and synthetic data-driven workflows and properties from a benchmark we create, bootstrapping from existing sets of business process specifications and properties by extending them with data-aware features. This is the first benchmark for business processes and properties that includes such data aware features. The experiments highlight the impact of the various optimizations and parameters of the workflows and the properties.

To further evaluate performance on practical workflows, we adapt to HAS* a standard complexity measure of control flow used in software engineering, *cyclomatic complexity* [128], and show experimentally, using the above benchmark, that cyclomatic complexity of HAS* specifications correlates meaningfully with verification times. Since conventional wisdom in software engineering holds that well-designed, human readable programs have relatively low cyclomatic complexity, this is an indication that verification times are likely to be good for well-designed HAS* specifications.

Taking this and other factors into account, the experimental results show that VERIFAS performs very well on practically relevant classes of data-driven workflows. Compared to SpinArt, it not only applies to a much broader class of data-driven workflows but also has a decisive performance advantage even on the simple workflows that SpinArt is able to handle. To our knowledge, VERIFAS is the first implementation of practical significance of a data-driven workflow verifier with full support for unbounded data.

The rest of the thesis is organized as follows. First, Chapter 2 presents a survey of the state-of-the-art of artifact verification. Chapter 3 introduces the HAS model with HLTL-FO and presents the decidability/complexity results. Chapter 4 describes the SpinArt implementation, with a discussion on the TAS model, a review of Spin and the optimization techniques for the Spin-based implementation. Chapter 5 presents VERIFAS, the specialized verifier built from scratch, including the HAS* model, various optimization techniques, benchmarks and experimental results. Finally, the thesis concludes with Chapter 6.

Chapter 2

Related Work

This chapter surveys the state-of-the-art of data-driven workflow verification. More specifically, we describe several models and results on verification, focusing on temporal properties of the underlying infinite-state transition systems. We review verification of business artifacts, and use it as a vehicle to introduce the main concepts and results. The technical challenges posed by verification of business artifacts are representative of those raised in some of the other models (notably data-driven web services), which can be viewed as syntactic variants of business artifacts. Section 2.8 summarizes some of the work pertaining specifically to data-driven web services.

2.1 Business Artifacts

IBM's business artifacts are a model of workflows in which data evolves under the action of "services" implementing business process tasks. They are a prominent exponent of data-aware business processes, that enrich the traditional process-centric approach by treating data as a first-class citizen. The notion of business artifact was first introduced in [103] and [85] (called there "adaptive documents"), and was further studied, from both practical and theoretical perspectives, in [14, 63, 64, 15, 92, 37, 76, 70]. Roots of the artifact model are present in "adaptive business objects" [101], "business entities", "document-driven" workflow [127] and "document" engineering [65]. The Vortex framework [78, 57, 77] also allows the specification of database manipulations and provides declarative specifications for when services are applicable to a given artifact.

The artifact model is inspired in part by the field of semantic web services. In particular, the OWL-S proposal [98, 95] describes the semantics of services in terms of input parameters, output parameters, pre- and post-conditions. In the artifact model considered here the services are applied in a sequential fashion (there is no true concurrency). IBM has developed Siena [35], a tool for compiling artifact-based procedural specifications into code supporting the corresponding business process. Its open-source descendant is the BizArtifact suite [19]. More recently, the Guard-Stage-Milestone (GSM) approach [37, 76] provides rich structuring mechanisms for services, including parallelism, concurrency and hierarchy, and has been incorporated in the OMG standard for Case Management Model and Notation (CMMN) [17, 94].

2.2 Tuple Artifact Systems

Early work on artifact verification focused on a minimalistic variant of the artifact model, called *tuple artifact system* (TAS). In a TAS, the artifact consists simply of a tuple of values that evolves throughout the workflow. TAS is described informally relying on an example (a more formal development can be found in Chapter 4 or [44, 36]). The example models an e-commerce business process in which the customer chooses a product and a shipment method and applies various kinds of coupons to the order. The artifact is an evolving tuple of values, referred to by variables (sometimes called *attributes*). The example has the following variables:

`status`, `prod_id`, `ship_type`, `coupon`, `amount_owed`, `amount_paid`, `amount_refunded`

The `status` variable tracks the status of the order and can take the following values:

“edit_product”, “edit_ship”, “edit_coupon” “processing”,

“received_payment”, “shipping”, “shipped”, “canceling”, “canceled”.

Artifact variables `ship_type` and `coupon` record the customer’s selection, received as an external input; `amount_paid` is also an external input (from the customer, possibly indirectly via a credit card service). Variable amount_owed is set by the system using arithmetic operations that sum up product price and shipment cost, subtracting the coupon value. Variable amount_refunded is set by the system in case a refund is activated.

The database is a finite first-order structure over a relational signature (called *schema* in database parlance), consisting of the following relations whose coordinates are given names, called *attributes*. Underlined attributes denote *keys*, which are attributes that uniquely identify each tuple in the relation.

- $\texttt{PRODUCTS}(\underline{\texttt{id}}, \texttt{price}, \texttt{availability}, \texttt{weight})$
- $\texttt{COUPONS}(\underline{\texttt{code}}, \texttt{type}, \texttt{value}, \texttt{min\_value}, \texttt{free\_shiptype})$
- $\texttt{SHIPPING}(\underline{\texttt{type}}, \texttt{cost}, \texttt{max\_weight})$
- $\texttt{OFFERS}(\underline{\texttt{prod\_id}}, \texttt{discounted\_price}, \texttt{active})$

The database also satisfies the following inclusions:

$$\texttt{COUPONS}[\texttt{free\_shiptype}] \subseteq \texttt{SHIPPING}[\texttt{type}]$$

$$\texttt{OFFERS}[\texttt{prod\_id}] \subseteq \texttt{PRODUCTS}[\texttt{id}]$$

The first inclusion says that each free_shiptype value in the COUPONS relation is also a type value in the SHIPPING relation. The second states that every prod_id value in the OFFERS is the actual id of a product in the PRODUCTS relation. In database terminology, it is said that free_shiptype and prod_id are *foreign keys*.

2.2.1 Services

Recall that artifacts evolve under the action of services. Each service is specified declaratively by a pre-condition π and a post-condition ψ, here limited to existential first-order ($\exists$FO) sentences. The pre-condition refers to the current values of the artifact variables and the database.

choose_product: The customer chooses a product.
π : $\texttt{status} =$ "edit_prod"
ψ : $\exists p, a, w(\texttt{PRODUCTS}(\texttt{prod\_id}', p, a, w) \wedge a > 0) \wedge \texttt{status}' =$ "edit_shiptype"
choose_shiptype: The customer chooses a shipping option.
π : $\texttt{status} =$ "edit_ship"
ψ : $\exists c, l, p, a, w(\texttt{SHIPPING}(\texttt{ship\_type}', c, l) \wedge \texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \wedge l > w) \wedge$
$\texttt{status}' =$ "edit_coupon" $\wedge\ \texttt{prod\_id}' = \texttt{prod\_id}$
apply_coupon: The customer optionally inputs a coupon number.
π : $\texttt{status} =$ "edit_coupon"
ψ : $(\texttt{coupon}' = \lambda \wedge \exists p, a, w, c, l(\texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \wedge$
$\texttt{SHIPPING}(\texttt{ship\_type}, c, l) \wedge \texttt{amount\_owed}' = p + c) \wedge \texttt{status}' =$ "processing"
$\wedge \texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type}) \vee$
$(\exists t, v, m, s, p, a, w, c, l(\texttt{COUPONS}(\texttt{coupon}', t, v, m, s) \wedge$
$\texttt{PRODUCTS}(\texttt{prod\_id}, p, a, w) \wedge \texttt{SHIPPING}(\texttt{ship\_type}, c, l) \wedge p + c \geq m \wedge$
$(t =$ "free_shipping" $\rightarrow (s = \texttt{ship\_type} \wedge \texttt{amount\_owed}' = p)) \wedge$
$(t =$ "discount" $\rightarrow \texttt{amount\_owed}' = p + c - v))$
$\wedge \texttt{status}' =$ "processing" $\wedge\ \texttt{prod\_id}' = \texttt{prod\_id} \wedge \texttt{ship\_type}' = \texttt{ship\_type})$

Figure 2.1. Three services

The post-condition ψ refers simultaneously to the current and *next* artifact values, as well as the database. In addition, both π and ψ may use arithmetic constraints on the variables, consisting of linear inequalities with integer coefficients.

Figure 2.1 shows some of the services for the business process of the example. The primed artifact variables x' refer to the *next* value of variable x.

Notice that the pre-conditions of the services check the value of the `status` variable. For instance, according to **choose_product**, the customer can only input her product choice while the order is in "edit_prod" status.

Also notice that the post-conditions constrain the next values of the artifact variables (denoted by a prime). For instance, according to **choose_product**, once a product has been picked, the next value of the status variable is "edit_shiptype", which will at a subsequent step enable the **choose_shiptype** service (by satisfying its pre-condition). Similarly, once the shipment type is chosen (as modeled by service **choose_shiptype**), the new status is "edit_coupon", which enables the **apply_coupon** service. The interplay of pre- and post-conditions achieves a sequential filling of the order, starting from the choice of product and ending with the claim of a coupon.

Notice the arithmetic computation used in the post-conditions. For instance, in service **apply_coupon**, the sum of the product price p and shipment cost c (looked up in the database) is adjusted with the coupon value (notice the distinct treatment of the two coupon types) and stored in the `amount_owed` artifact variable.

2.2.2 Semantics

The semantics of a TAS $\mathcal{A}$ consists of its *runs*. Given a database D, a run of $\mathcal{A}$ is an infinite sequence $\{\rho_i\}_{\geq 0}$ of artifact tuples such that ρ_0 and D satisfy the initial condition of the system, and for each $i \geq 0$ there is a service S of the system such that ρ_i and D satisfy the pre-condition of S and ρ_i, ρ_{i+1} and D satisfy its post-condition. For uniformity, blocking prefixes of runs are extended to infinite runs by repeating forever their last tuple.

Note that the above semantics only considers linear runs of the system. A more informative notion is the *tree of runs* that completely captures the choice of services applicable at any given stage in the computation. Formulating and verifying branching-time properties would require a semantics consisting of the full tree of runs.

2.3 Specifying Temporal Properties

Verifying temporal properties of runs of data-driven workflows is of interest in this thesis. For instance, in the artifact system example, the following property is desired to be verified:

- If a correct payment is submitted then at some time in the future either the product is shipped or the customer is refunded the correct amount.

An extension of LTL (linear-time temporal logic) is used in order to specify such temporal properties. Recall that LTL is propositional logic augmented with temporal operators such as **G** (always), **F** (eventually), **X** (next) and **U** (until) (e.g., see [105]). For example, $\mathbf{G}p$ says that p holds at all times in the run, $\mathbf{F}p$ says that p will eventually hold, and $\mathbf{G}(p \rightarrow \mathbf{F}q)$ says that whenever p holds, q must hold sometime in the future. The extension of LTL, called[1] LTL-FO, is obtained from LTL by replacing propositions with quantifier-free FO statements about particular artifact tuples in the run. The statements use the artifact variables and may use additional *global* variables, shared by different statements and allowing to refer to values in different tuples. The global variables are universally quantified over the entire property.

For example, to specify the above example property, the LTL-FO formula is of the form $\mathbf{G}(p \rightarrow \mathbf{F}q)$, where p says that a correct payment is submitted and q states that either the product is shipped or the customer is refunded the correct amount. Moreover, if the customer is refunded, the amount of the correct payment (given in p) should be the same as the amount of the refund (given in q). This requires using a global variable x in both p and q. More precisely, p is interpreted as the formula $\texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}$ and q as $\texttt{status} = \text{"shipped"} \vee \texttt{amount\_refunded} = x$. This yields the LTL-FO property

$$(\varphi) \qquad \forall x\, \mathbf{G}((\texttt{amount\_paid} = x \wedge \texttt{amount\_paid} = \texttt{amount\_owed}) \\ \rightarrow \mathbf{F}(\texttt{status} = \text{"shipped"} \vee \texttt{amount\_refunded} = x))$$

Note that, as one would expect, the global variable x is universally quantified at the outermost scope. An artifact system $\mathcal{A}$ satisfies an LTL-FO sentence φ if all runs of the artifact system satisfy φ for all values of the global variables. Note that the database is fixed for each run, but may be different for different runs.

Note that variants of LTL-FO have been introduced in [58, 120]. The use of globally quantified variables is also similar in spirit to the *freeze quantifier* defined in the context of LTL extensions with data by Demri and Lazić [42, 43].

[1]The variant of LTL-FO used here differs from previous ones in that the FO formulas interpreting propositions are quantifier-free. By slight abuse the same name is used in this thesis.

2.4 Automatic Verification of Tuple Artifact Systems

Classical model checking applies to finite-state transition systems. While finite-state systems may fully capture the semantics of some systems to be verified (for example logical circuits), most software systems are in fact infinite-state systems, of which a finite-state transition system represents a rough abstraction. Properties of the actual system are also abstracted, using a finite set of propositions whose truth values describe each of the finite states of the transition system. Checking that an LTL property holds is done by searching for a counterexample run of the system. Its finiteness is essential and allows to decide property satisfaction in PSPACE using an automata-theoretic approach (see e.g. [34, 99]).

Consider now a TAS $\mathcal{A}$ and an LTL-FO property φ. Model checking $\mathcal{A}$ with respect to φ can be viewed once again as a search for a counterexample run of $\mathcal{A}$, i.e. a run violating φ. The immediate difficulty, compared to the classical approach, stems from the fact that $\mathcal{T}_{\mathcal{A}}$ is an infinite-state system. To obtain decidability in this context, the typical approach consists of using *symbolic representations* of runs, as described later.

In the broader context of verification, research on automatic verification of infinite-state systems has also focused on extending classical model checking techniques (e.g., see [28] for a survey). However, in much of this work the emphasis is on studying recursive control rather than data, which is either ignored or finitely abstracted. More recent work has been focusing specifically on data as a source of infinity. This includes augmenting recursive procedures with integer parameters [22], rewriting systems with data [23, 21], Petri nets with data associated to tokens [86], automata and logics over infinite alphabets [25, 24, 102, 42, 81, 20, 21], and temporal logics manipulating data [42, 43]. However, the restricted use of data and the particular properties verified have limited applicability to the business artifact setting, or other database-driven applications.

2.5 Tuple artifacts without constraints or dependencies

First, tuple artifact systems and properties without arithmetic constraints or data dependencies are considered. This case was studied in [44], with a slightly richer model in which artifacts can carry some limited relational state information (however, here the discussion sticks for simplicity to the earlier minimalistic model). The main result is the following.

Theorem 1. *It is decidable, given a TAS $\mathcal{A}$ with no data dependencies or arithmetic constraints, and an LTL-FO property φ with no arithmetic constraints, whether $\mathcal{A}$ satisfies φ.*

The complexity of verification is PSPACE-complete for fixed-arity database, and EXPSPACE otherwise. This is the best one can expect, given that even very simple static analysis problems for finite-state systems are already PSPACE-complete [117].

The main idea behind the verification algorithm is to explore the space of runs of the artifact system using *symbolic* runs rather than actual runs. This is based on the fact that the relevant information at each instant is the pattern of connections in the database between attribute values of the current and successor artifact tuples in the run, referred to as their *isomorphism type*. Indeed, the sequence of isomorphism types in a run can be generated symbolically and is enough to determine satisfaction of the property. Since each isomorphism type can be represented by a polynomial number of tuples (for fixed arity), this yields PSPACE verification.

It turns out that the verification algorithm can be extended to specifications and properties that use a *total order* on the data domain, which is useful in many cases. This however complicates the algorithm considerably, since the order imposes global constraints that are not captured by the local isomorphism types. The algorithm was first extended in [44] for the case of a dense countable order with no end-points. This was later generalized to an arbitrary total order by Segoufin and Torunczyk [111] using automata-theoretic techniques. In both cases, the worst-case complexity remains PSPACE.

2.6 Tuple artifacts with arithmetic constraints and data dependencies

Unfortunately, Theorem 1 fails even in the presence of simple data dependencies or arithmetic. Specifically, as shown in [44, 36], verification becomes undecidable as soon as the database is equipped with at least one key dependency, *or* if the specification of the artifact system uses simple arithmetic constraints allowing to increment and decrement by one the value of some attributes. Hence, a restriction is needed to achieve decidability as discussed next.

To gain some intuition, consider the undecidability of verification for TAS with increments and decrements. The proof of undecidability is based on the ability of such systems to simulate *counter machines*, for which the problem of state reachability is known to be undecidable [100]. To simulate counter machines, a TAS uses an attribute for each counter. A service performs an increment (or decrement) operation by "feeding back" the incremented (or decremented) value into the next occurrence of the corresponding attribute. To simulate counters, this must be done an unbounded number of times. To prevent such computations, the restriction imposed in [36] is designed to limit the data flow between occurrences of the same artifact attribute at different times in runs of the system that satisfy the desired property. As a first cut, a possible restriction would prevent any data flow path between unequal occurrences of the same artifact attribute. Let us call this restriction *acyclicity*. While acyclicity would achieve the goal of rendering verification decidable, it is too strong for many practical situations. In the running example, a customer can choose a shipping type and coupon and repeatedly change her mind and start over. Such repeated performance of a task is useful in many scenarios, but would be prohibited by acyclicity of the data flow. To this end, a more permissive restriction called *feedback freedom* is defined in [36]. The formal definition considers, for each run, a graph capturing the data flow among variables, and imposes a restriction on the graph. Intuitively, paths among different occurrences of the same attribute are permitted, but only as long as each value of the attribute is independent on its previous values. This is ensured by a syntactic condition that takes into account both the TAS and the property to be verified. It is shown in [36] that feedback freedom of a TAS together with an LTL-FO property can be checked in PSPACE by reduction to a test of emptiness of a two-way alternating finite-state automaton. Feedback freedom turns out to ensure decidability of verification in the presence of arithmetic constraints, and also under a large class of data dependencies including key and foreign key constraints on the database.

Theorem 2. *[36] It is decidable, given a TAS $\mathcal{A}$ whose database satisfies a set of key and foreign key constraints, and an LTL-FO property φ such that $(\mathcal{A}, \varphi)$ is feedback free, whether every run of $\mathcal{A}$ on a valid database satisfies φ.*

The intuition behind decidability is the following. Recall the verification algorithm of Theorem 1. Because of the data dependencies and arithmetic constraints, the isomorphism types of symbolic runs no longer suffice, because every artifact tuple in a run is constrained by the entire history leading up to it. This can be specified as an $\exists$FO formula using one quantified variable for each artifact attribute occurring in the history, referred to as the *inherited constraint* of the tuples. The key observation is that due to feedback freedom, the inherited constraint can be rewritten into an $\exists$FO formula with quantifier rank bounded by k^2, where k is the number of attributes of the artifact (the quantifier rank of a formula is the maximum number of quantifiers occurring along a path from root to leaf in the syntax tree of the formula, see [90]). This implies that there are only finitely many non-equivalent inherited constraints. This allows to use again a symbolic run approach to verification, by replacing isomorphism types with inherited constraints. However, the complexity of the resulting algorithm is non-elementary (a tower of exponentials of height k^2).

One might wonder if the decidability results of this section can be extended to branching-time logics (CTL or CTL*). Unfortunately, it is easily shown that even very simple CTL properties become undecidable in the above framework. It remains open whether there are reasonable restrictions that guarantee decidability of CTL or CTL*. Limited positive results on verification of branching-time properties of data-driven web services are obtained in [49].

2.7 Other work on verification of artifact systems

Initial work on formal analysis of artifact-based business processes in restricted contexts has investigated reachability [63, 64], general temporal constraints [64], and the existence of complete execution or dead end [15]. For each considered problem, verification is generally undecidable; decidability results were obtained only under rather severe restrictions, e.g., restricting all pre-conditions to be "true" [63], restricting to bounded domains [64, 15], or restricting the pre- and post-conditions to be propositional, and thus not referring to data values [64]. [29] adopts an artifact model variation with arithmetic operations but no database. Decidability relies on restricting runs to bounded length. [129] addresses the problem of the existence of a run that satisfies a temporal property, for a restricted case with no database and only propositional LTL properties. None of these works model an underlying database, artifact relations, task hierarchy, or arithmetic.

A more recent line of work has tackled the verification of artifact systems in which properties are checked only over the runs starting from a given initial database that may evolve via updates, insertions and deletions. [10, 9, 11, 12, 38] consider several models and property languages, culminating in [71], which addresses verification of first-order μ-calculus (hence branching time) properties in a framework that is equivalent to artifact systems whose input is provided by external services. [13, 31] extend the results of [71] to artifact-centric multi-agent systems where the property language is a version of first-order branching-time temporal-epistemic logic expressing the knowledge of the agents. This line of work uses variations of a business process model called DCDS (data-centric dynamic systems), which is sufficiently expressive to capture the GSM model, as shown in [119]. In their unrestricted form, DCDS and HAS (introduced in Chapter 3) have similar expressive power. However, verification for DCDS is only considered for a *fixed* rather than arbitrary initial database. Recently, [4] considered verification of monadic second-order properties of runs in a model where the underlying database can be updated by insertions and deletions. Decidability is obtained subject to a restriction called

k-recency boundedness, allowing only the most recent k elements in the database to be modified by an update, for a fixed k.

See [80] for a survey on data-centric business process management, and [30] for a survey of corresponding verification results.

2.7.1 Verifier implementations for data-driven workflows

On the practical side of data-driven workflow verification, [41] considers the verification of business processes specified in a Petri-net-based model extended with data and process components, in the spirit of the theoretical work of [110, 6, 87, 112], which considers extending Petri nets with data-carrying tokens. The verifier of [41] checks properties for a given initial database. [68] and its prior work [66, 67] implemented a verifier for artifact systems specified directly in the GSM model. While the above models are expressive, the verifiers require restrictions of the models strongly limiting modeling power [66], or predicate abstraction resulting in loss of soundness and/or completeness [67, 68]. Lastly, the properties verified in [67, 68] focus on temporal-epistemic properties in a multi-agent finite-state system.

Practical verification has also been studied in business process management (see [123] for a survey). The considered models are mostly process-driven (BPMN, Workflow-Net, UML etc.), with the business-relevant data abstracted away. The implementation of a verifier for data-driven web applications was studied in [46, 50]. The model is similar in flavor to the artifact model, but much less expressive. The verification approach developed there is not applicable to the models introduced in this thesis, which requires substantially new tools and techniques.

2.8 Data-Driven Web Services

The goal of the Web services paradigm is to enable the use of Web-hosted services with a high degree of flexibility and reliability. Web services can function in a stand-alone manner, or they can be “glued” together into multi-peer *compositions* that implement complex applications. To describe and reason about Web services, various standards and models have been proposed, focusing on different levels of abstraction and targeting different aspects of the Web service (see [79] for a tutorial).

A commercially successful high-level specification tool for web applications is Web Ratio [2], an outgrowth of the earlier academic prototype WebML [32, 26]. This section illustrates with an example the WebML approach to specifying data-driven web services, formally studied in [48, 49]. Consider the common scenario of a web service that takes input from external users and responds by producing output. The contents of a Web page is determined dynamically by querying the underlying database as well as the state. The output of the Web site, transitions from one Web page to another, and state updates, are determined by the current input, state, and database, and defined by first-order queries. Figure 2.2 illustrates a WebML-style specification of an e-commerce Web site selling computers online. New customers can register a name and password, while returning customers can login, search for computers fulfilling certain criteria, add the results to a shopping cart, and finally buy the items in the shopping cart.

A run of the above Web site starts as follows. Customers begin at the home page by providing their login name and password, and choosing one of the provided buttons (login, register, or cancel). Suppose the choice is to login. The reaction of the Web site is determined by a query checking if the name and password provided are found in the database of registered users. If the answer is positive, the login is successful and the customer proceeds to the Customer page or the Administration page depending on his status. Otherwise, there is a transition to the Error page. This continues as described by the flowchart in the figure.

2.8.1 Verification of data-driven web services

The verification problem for database-driven web services has been studied using a transducer-based formal model, called *Extended Abstract State Machine Transducer*, in brief ASM$^+$. The model is an extension of the Abstract State Machine (ASM) transducer previously studied by Spielmann [120]. Similarly to the earlier Relational Transducer [5], the ASM$^+$ transducer models database-driven reactive systems that respond to input events by producing

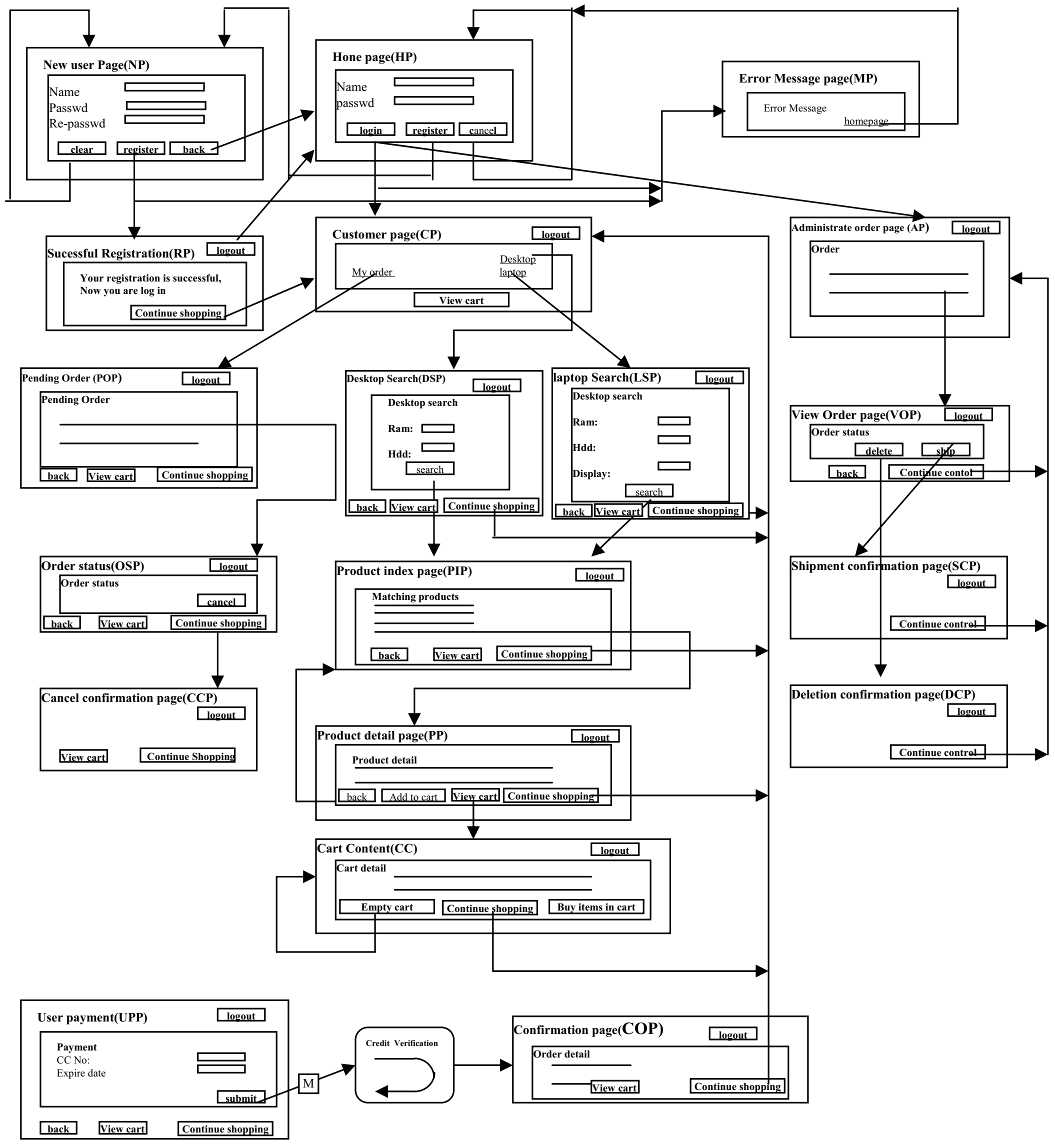


Figure 2.2. Web pages in the computer shopping site.

some output, and maintain state information in designated relations. The control of the device is specified using first-order queries. The main motivation for ASM$^+$ transducers is that they are sufficiently powerful to simulate complex Web service specifications in the style of WebML. Thus, they are a convenient vehicle for developing the theoretical foundation for the verification of such systems, and they also provide the basis for the implementation of a verifier.

As in the case of business artifacts, restrictions are needed on the ASM$^+$ transducers and properties in order to ensure decidability of verification. The main restriction, first proposed in [120] for ASM transducers, is called "input boundedness". The core idea of input boundedness is that quantifications used in formulas of the specification and property are guarded by input atoms. For example, if *pay* is an input, the LTL-FO formula (where **B** is shorthand for *before*)

$$\forall x\ (\mathbf{G}\ (\exists z(pay(x,z) \wedge price(x,z))\ \mathbf{B}\ ship(x)))$$

is input bounded, since the quantification $\exists z$ is guarded by $pay(x,z)$. This restriction matches naturally the intuition that the system modeled by the transducer is input driven. The actual restriction is quite technical, but provides an appealing package. First, it turns out to be tight, in the sense that even small relaxations lead to undecidability. Second, as argued in [48, 49], it remains sufficiently rich to express a significant class of practically relevant applications and properties. As a typical example, the e-commerce Web application illustrated in Figure 2.2 can be modeled under this restriction, and many relevant natural properties can be expressed. Third, as in the case of tuple artifacts without dependencies or arithmetic, the complexity of verification is PSPACE (for fixed-arity schemas). Moreover, the proof technique developed to show decidability in PSPACE provides the basis for the implementation of an actual verifier, described in Section 2.8.3.

2.8.2 Compositions of ASM$^+$ Transducers

The verification results discussed above apply to single ASM$^+$ transducers in isolation. These results were extended in [52] to the more challenging case of *compositions* of ASM$^+$ transducers, modeling compositions of database-driven Web services. Asynchronous communication between transducers adds another dimension that has to be taken into account. In an ASM$^+$ composition, the transducers communicate with each other by sending and receiving messages via one-way channels. Properties of runs to be verified are specified in an extension of LTL-FO, where the FO components may additionally refer to the messages currently read and received.

Towards decidable verification, the input-boundedness restriction is extended in the natural way. Additional restrictions must be placed on the message channels: they may be lossy, but are required to be bounded. With these restrictions, verification is again shown to be PSPACE-complete (for fixed-arity relations, and EXPSPACE otherwise). The proof is by reduction to the single transducer case, and the restrictions are shown to be tight.

As in the case of single transducers, verification becomes undecidable if some of the restrictions are relaxed. Not surprisingly, verification is undecidable with unbounded queues (this already happens for finite-state systems [27]). More interestingly, lossiness of channels is essential: verification becomes undecidable under the assumption that channels are *perfect*, i.e. messages are never lost (the proof is by reduction of the Post Correspondence Problem [106]).

The above model of compositions assumes that all specifications of participating peers are available to the verifier. However, compositions may also involve autonomous parties unwilling to disclose the internal implementation details. In this case, the only information available is typically a specification of their input-output behavior. This leads to an investigation of *modular* verification. It consists in verifying that a subset of fully specified transducers behaves correctly, subject to input-output properties of the other transducers. Decidability results are obtained in [52] for modular verification, subject to an appropriate extension of the input-boundedness restriction.

2.8.3 The WAVE Verifier

While the PSPACE upper bound obtained for verification of ASM$^+$ transducers in the input-bounded case is encouraging from a theoretical viewpoint, it does not provide any indication of practical feasibility. Fortunately, it turns out that the symbolic approach described above also provides a good basis for efficient implementation. Indeed, this technique lies at the core of the WAVE verifier, targeted at data-driven Web services of the WebML flavor [52, 47].

The verifier, as well as its target specification framework, are both implemented from scratch. First, a tool is developed for high-level, efficient specification of data-driven Web services, in the spirit of WebML. Next, WAVE is implemented taking as input a specification of a Web service using the tool, and an LTL-FO property to be verified. The starting point for the implementation is the symbolic run technique. The verifier basically carries out a search for counterexample symbolic runs. However, verification becomes practical only in conjunction with an array of additional heuristics and optimization techniques, yielding critical improvements. Chief among these is dataflow analysis, allowing to dramatically prune the search for counterexample symbolic runs.

The verifier was evaluated on a set of practically significant Web application specifications, mimicking the core features of sites such as Dell, Expedia, and Barnes and Noble. The experimental results show very good verification times (on the order of seconds), suggesting that automatic verification is practically feasible for significant classes of properties and Web services. The implementation and experimental results are described in [47], and a demo of the WAVE prototype was presented in [51].

Chapter 3

Verification of Hierarchical Artifact Systems

3.1 Overview

As discussed in Chapter 2, in previous work [44, 36], the verification problem was studied for a bare-bones variant of artifact systems, without hierarchy or concurrency, in which each artifact consists of a flat tuple of evolving values and the services are specified by simple pre-and-post conditions on the artifact and database. More precisely, the problem considered was to statically check whether all runs of an artifact system satisfy desirable properties expressed in LTL-FO, an extension of linear-time temporal logic where propositions are interpreted as ∃FO sentences on the database and current artifact tuple. In order to deal with the resulting infinite-state system, in [44], a symbolic approach was developed to allow a reduction to finite-state model checking and yielding a PSPACE verification algorithm for the simplest variant of the model (no database dependencies and uninterpreted data domain). In [36] the approach was extended to allow for database dependencies and numeric data testable by arithmetic constraints. Unfortunately, decidability was obtained subject to a rather complex semantic restriction on the artifact system and property (feedback freedom), and the verification algorithm has non-elementary complexity.

The present chapter describes a significant advance on the artifact verification problem on several fronts. We consider a much richer and more realistic model, called *Hierarchical Artifact*

System (HAS), abstracting core elements of the GSM model. In particular, the model features task hierarchy, concurrency, and richer artifact data (including updatable artifact relations). We consider properties expressed in a novel *hierarchical* temporal logic, HLTL-FO, that is well-suited to the model. Our main results establish the complexity of checking HLTL-FO properties for various classes of HAS, highlighting the impact of various features on verification. The results require qualitatively novel techniques, because the reduction to finite-state model checking used in previous work is no longer possible. Instead, the richer model requires the use of a hierarchy of Vector Addition Systems with States (VASS) [18]. The arithmetic constraints are handled using quantifier elimination techniques, adapted to our setting.

We next describe the model and results in more detail. A HAS consists of a database and a hierarchy (rooted tree) of *tasks*. Each task has associated to it local evolving data consisting of a tuple of artifact variables and an updatable artifact relation. It also has an associated set of *services*. Each application of a service is guarded by a pre-condition on the database and local data and causes an update of the local data, specified by a post condition (constraining the next artifact tuple) and an insertion or retrieval of a tuple from the artifact relation. In addition, a task may invoke a child task with a tuple of parameters, and receive back a result if the child task completes. A run of the artifact system consists of an infinite sequence of transitions obtained by any valid interleaving of concurrently running task services.

In order to express properties of HAS's we introduce a subset of LTL-FO called *hierarchical* LTL-FO (HLTL-FO). Intuitively, an HLTL-FO formula uses as building blocks LTL-FO formulas acting on runs of individual tasks, called local runs, referring only to the database and local data, and can recursively state HLTL-FO properties on runs resulting from calls to children tasks. The language HLTL-FO closely fits the computational model and is also motivated on technical grounds discussed in the paper. A main justification for adopting HLTL-FO is that LTL-FO (and even LTL) properties are undecidable for HAS's.

Hierarchical artifact systems as sketched above provide powerful extensions to the variants previously studied, each of which immediately leads to undecidability of verification if not carefully controlled. Our main contribution is to put forward a package of restrictions that ensures decidability while capturing a significant subset of the GSM model. This requires a delicate balancing act aiming to limit the dangerous features while retaining their most useful aspects. In contrast to [36], this is achieved without the need for unpleasant semantic constraints such as feedback freedom. The restrictions are discussed in detail in the paper, and shown to be necessary by undecidability results.

The complexity of verification under various restrictions is summarized in Tables 3.1 (without arithmetic) and 3.2 (with arithmetic). As seen, the complexity ranges from PSPACE to non-elementary for various packages of features. The non-elementary complexity (a tower of exponentials whose height is the depth of the hierarchy) is reached for HAS with cyclic schemas, artifact relations and arithmetic. For acyclic schemas, which include the widely used Star (or Snowflake) schemas [84, 126], the complexity ranges from PSPACE (without arithmetic or artifact relations) to double-exponential space (with both arithmetic and artifact relations). This is a significant improvement over the previous algorithm of [36], which even for acyclic schemas has non-elementary complexity in the presence of arithmetic (a tower of exponentials whose height is the square of the total number of artifact variables in the system).

This chapter is organized as follows. The HAS model is presented in Section 3.2. We present its syntax and semantics, including a representation of runs as a tree of local task runs, that factors out interleavings of independent concurrent tasks. The temporal logic HLTL-FO is introduced in Section 3.3, together with a corresponding extension of Büchi automata to trees of local runs. Section 3.4 justifies the restrictions imposed on the HAS model by showing that lifting any of them leads to undecidability of verification. In Section 3.5 we prove the decidability of verification for HAS without arithmetic, and establish its complexity. To this end, we develop a symbolic representation of HAS runs and a reduction of model checking to state reachability problems in a set of nested VASS (mirroring the task hierarchy). In Section 3.6 we show how the verification results can be extended in the presence of arithmetic. Finally, we conclude in Section 3.7. The appendix provides more details and proofs.

3.2 Framework

In this section, we present the syntax and semantics of Hierarchical Artifact Systems (HAS's). The formal definitions are illustrated with an intuitive example of the HAS specification of a travel booking business process inspired by Expedia [60]. At a high level, the example captures a process where a customer books flights and/or makes hotel reservations.

3.2.1 Syntax of HAS

We begin with the underlying database schema. We assume familiarity with the notions of *key* and *foreign key* (e.g., see [114]), as well as first-order formula (FO), existential FO ($\exists$FO), and quantifier-free FO (e.g., see [90]).

Definition 3. *A **database schema** $\mathcal{DB}$ is a finite set of relation symbols, where each relation R of $\mathcal{DB}$ has an associated sequence of distinct attributes containing the following:*

- *a key attribute* ID *(present in all relations),*
- *a set of foreign key attributes $\{F_1, \ldots, F_m\}$, and*
- *a set of non-key attributes $\{A_1, \ldots, A_n\}$ disjoint from $\{\text{ID}, F_1, \ldots, F_m\}$.*

To each foreign key attribute F_i of R is associated a relation R_i of $\mathcal{DB}$ and the inclusion dependency $R[F_i] \subseteq R_i[\text{ID}]$ (stating that the projection of R on F_i is included in the projection of R_i on ID*). It is said that F_i references R_i. We denote by $attr(R)$ the set of attributes of R.*

The domain $Dom(A)$ of each attribute A depends on its type. The domain of all non-key attributes is numeric, specifically $\mathbb{R}$. The domain of each key attribute is a countable infinite domain disjoint from $\mathbb{R}$. For distinct relations R and R', $Dom(R.\textit{ID}) \cap Dom(R'.\textit{ID}) = \emptyset$. The domain of a foreign key attribute F referencing R is $Dom(R.\textit{ID})$. We denote by $\textit{DOM}_{id} = \cup_{R \in \mathcal{DB}} Dom(R.\textit{ID})$. Intuitively, in such a database schema, each tuple is an object with a *globally* unique id. This id does not appear anywhere else in the database except in foreign keys referencing it. An *instance* of a database schema $\mathcal{DB}$ is a mapping D associating to each relation symbol R a finite relation $D(R)$ of the same arity as R, whose tuples provide, for each attribute, a value from its domain. In addition, D satisfies all key and inclusion dependencies associated with the keys and foreign keys of the schema. The active domain of D, denoted $\texttt{adom}(D)$, consists of all elements of D (id's and reals). A database schema $\mathcal{DB}$ is *acyclic* if there are no cycles in the references induced by foreign keys. More precisely, consider the labeled graph FK whose nodes are the relations of the schema and in which there is an edge from R_i to R_j labeled with F if R_i has a foreign key attribute F referencing R_j. The schema $\mathcal{DB}$ is *acyclic* [1] if the graph FK is acyclic, and it is *linearly-cyclic* if each relation R is contained in at most one simple cycle. A main reason for considering these special schemas is that they lead to significantly improved complexity of verification. The most restricted, acyclic schemas, still capture Star and Snowflake schemas [84, 126], widely used in storing business process data.

Example 4. *The HAS of the travel booking business process has the following database schema:*

- $\texttt{FLIGHTS}(\texttt{ID}, \texttt{price}, \texttt{comp\_hotel\_id})$

  $\texttt{HOTELS}(\texttt{ID}, \texttt{unit\_price}, \texttt{discount\_price})$

In the schema, the `ID`*'s are key attributes,* `price`, `unit_price`, `discount_price` *are non-key attributes, and* `comp_hotel_id` *is a foreign key attribute satisfying the inclusion dependency:*

$$\texttt{FLIGHTS}[comp\_hotel\_id] \subseteq \texttt{HOTELS}[\texttt{ID}].$$

Intuitively, each flight stored in the `FLIGHTS` *table has a hotel compatible for discount. If a flight is purchased together with a compatible hotel reservation, a discount is applied on the hotel reservation. Otherwise, the full price needs to be paid. The schema is acyclic, since* $\texttt{FLIGHTS}[comp\_hotel\_id] \subseteq \texttt{HOTELS}[\texttt{ID}]$ *is the only inclusion dependency in the schema.*

The assumption that the ID of each relation is a single attribute is made for simplicity, and multiple-attribute IDs can be easily handled. The fact that the domain of all non-key attributes

[1] Here a cycle is a sequence of relations $\{R_i\}_{1\leq i\leq k}$ where $k \geq 2$, $R_1 = R_k$ and there is a foreign key reference from R_{i-1} to R_i for every $i > 1$.

is numeric is also harmless. Indeed, a uninterpreted domain on which only equality can be used can be easily simulated. Note that the keys and foreign keys used on our schemas are special cases of the dependencies used in [36]. The limitation to keys and foreign keys is one of the factors leading to improved complexity of verification and still captures most schemas of practical interest.

We next proceed with the definition of tasks and services, described informally in the introduction. The definition imposes various restrictions needed for decidability of verification. These are discussed and motivated in Section 3.4.

Similarly to the database schema, we consider two infinite, disjoint sets VAR_{id} of ID variables and VAR_{val} of numeric variables. We associate to each variable x its *domain* $Dom(x)$. If $x \in \mathit{VAR}_{id}$, then $Dom(x) = \{\texttt{null}\} \cup \mathit{DOM}_{id}$, where null $\notin \mathit{DOM}_{id} \cup \mathbb{R}$ (null plays a special role that will become clear shortly). If $x \in \mathit{VAR}_{val}$, then $Dom(x) = \mathbb{R}$. An *artifact variable* is a variable in $\mathit{VAR}_{id} \cup \mathit{VAR}_{val}$. If $\bar{x}$ is a sequence of artifact variables, a *valuation* of $\bar{x}$ is a mapping ν associating to each variable in $\bar{x}$ an element of its domain $Dom(x)$.

Definition 5. *A* ***task schema*** *over database schema $\mathcal{DB}$ is a triple $T = \langle \bar{x}^T, S^T, \bar{s}^T, \bar{x}^T_{\mathsf{in}}, \bar{x}^T_{\mathsf{out}} \rangle$ where $\bar{x}^T$ is a sequence of distinct artifact variables, S^T is a relation symbol not in $\mathcal{DB}$ with associated arity k, $\bar{s}^T$ is a sequence of k distinct ID variables in $\bar{x}^T$ and $\bar{x}^T_{\mathsf{in}}$ and $\bar{x}^T_{\mathsf{out}}$ are subsequences of $\bar{x}^T$ called the input and output variables of T.*

We denote by $\bar{x}^T_{id} = \bar{x}^T \cap \mathit{VAR}_{id}$ and $\bar{x}^T_{\mathbb{R}} = \bar{x}^T \cap \mathit{VAR}_{val}$. We refer to S^T as the *artifact relation* or *set* of T. Intuitively, an artifact relation is an updatable set where a task can insert/retrieve tuples. As we shall see, tuples of artifact relations are restricted to contain values from DOM_{id}. The data stored in the artifact variables and relations together represent the current state of a task.

Example 6. ***ManageTrips*** *is a task in the travel booking artifact system. This task models the process whereby the customer creates, stores, and retrieves candidate trips. A trip consists of a flight and/or hotel reservation. The customer can also choose one of the candidate trips and* *finalize the booking. The task has the following artifact variables:*

- *ID variables:* `flight_id`, `hotel_id`,
- *numeric variables:* `amount_paid`, `status`.

It also has an artifact relation `TRIPS` *storing candidate trips* (`flight_id`, `hotel_id`). ***ManageTrips*** *has no input/output variables.*

Definition 7. *An **artifact schema** is a tuple $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$ where $\mathcal{DB}$ is a database schema and $\mathcal{H}$ is a rooted tree of task schemas over $\mathcal{DB}$ with pairwise disjoint sets of artifact variables*[2] *and distinct artifact relation symbols.*

The rooted tree $\mathcal{H}$ defines the *task hierarchy*. Suppose the set of tasks is $\{T_1, \ldots, T_k\}$. For uniformity, we always take task T_1 to be the root of $\mathcal{H}$. We denote by $\preceq_{\mathcal{H}}$ (or simply $\preceq$ when $\mathcal{H}$ is understood) the partial order on $\{T_1, \ldots, T_k\}$ induced by $\mathcal{H}$ (with T_1 the minimum). For a node T of $\mathcal{H}$, we denote by *tree(T)* the subtree of $\mathcal{H}$ rooted at T, *child*(T) the set of children of T (also called *subtasks* of T), *desc*(T) the set of descendants of T (excluding T). Finally, *desc*$^*(T)$ denotes *desc*$(T) \cup \{T\}$. We denote by $\mathcal{S}_{\mathcal{H}}$ (or simply $\mathcal{S}$ when $\mathcal{H}$ is understood) the relational schema $\{S^{T_i} \mid 1 \leq i \leq k\}$. An instance of $\mathcal{S}$ is a mapping associating to each $S^{T_i} \in \mathcal{S}$ a finite relation over DOM_{id} of the same arity.

Example 8. *The travel booking artifact system has the following 4 tasks: T_1:**ManageTrips**, T_2:**AddHotel**, T_3:**AddFlight** and T_4:**BookTrip**, which form the hierarchy represented in Fig. 5.1.*

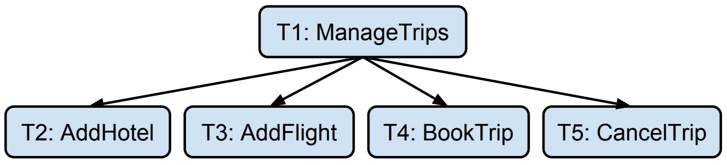


Figure 3.1. Tasks hierarchy.

*The process implemented by the above tasks can be described informally as follows. At the root task **ManageTrips**, the customer can add a flight and/or hotel to the trip by calling the*

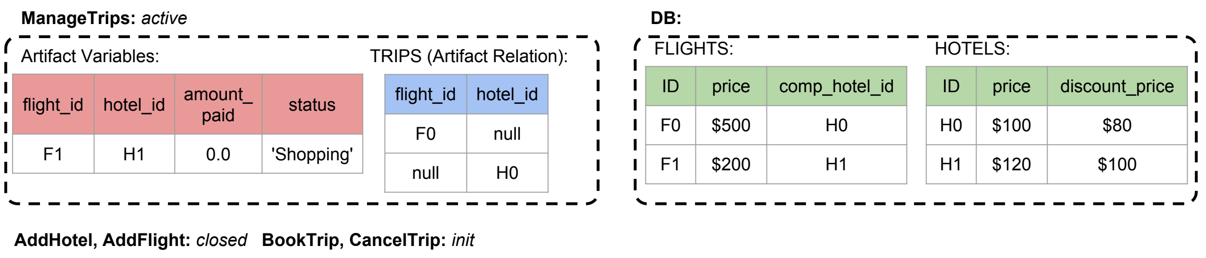
[2]In examples we sometimes use for convenience the same artifact variable names in several tasks, with the understanding that they represent distinct variables.

AddHotel *or the* ***AddFlight*** *tasks. The customer can also store candidate trips in the artifact relation* TRIPS *and retrieve previously stored trips. After the customer has made a decision, the* ***BookTrip*** *task is called to book the trip and the payment is processed. After the payment, the customer can decide to cancel the flight and/or the hotel reservation using the* ***CancelTrip*** *task and receive a refund.*

Definition 9. *An* ***instance*** *of an artifact schema $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$ is a tuple $\bar{I} = \langle \bar{\nu}, stg, D, \bar{S} \rangle$ where D is an instance of $\mathcal{DB}$, $\bar{S}$ an instance of $\mathcal{S}$, $\bar{\nu}$ a valuation of $\bigcup_{i=1}^{k} \bar{x}^{T_i}$, and stg (standing for "stage") a mapping of $\{T_1, \ldots, T_k\}$ to $\{\texttt{init}, \texttt{active}, \texttt{inactive}\}$.*

The stage $stg(T_i)$ of a task T_i has the following intuitive meaning in the context of a run of its parent: `init` indicates that T_i is inactive and available to be called, `active` says that T_i has been called and has not yet returned its answer, and `inactive` indicates that T_i has returned its answer. As we shall see, T_i cannot be called while its stage is `inactive`, but the model provides a way to reset the stage to `init`. Thus, T_i can be called multiple times during a run of its parent. However, only one instance of T_i can be active at any given time.

Example 10. *Fig. 5.2 shows an example of an instance of the travel booking business process specified in HAS. The only active task is* ***ManageTrip****.*


ManageTrips: active
Artifact Variables:
TRIPS (Artifact Relation):
DB:
FLIGHTS:
HOTELS:
AddHotel, AddFlight: closed BookTrip, CancelTrip: init

flight_id	hotel_id	amount_ paid	status
F1	H1	0.0	'Shopping'

flight_id	hotel_id
F0	null
null	H0

ID	price	comp_hotel_id
F0	$500	H0
F1	$200	H1

ID	price	discount_price
H0	$100	$80
H1	$120	$100

Figure 3.2. An instance of the travel booking schema.

We proceed with the definition of *conditions*, used to specify task services. We denote by $\mathcal{C}$ an infinite set of relation symbols, each of which has a fixed interpretation as the set of real solutions of a finite set of polynomial inequalities with integer coefficients. By slight abuse, we sometimes use the same notation for a relation symbol in $\mathcal{C}$ and its fixed interpretation. For a given artifact schema $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$ and a sequence $\bar{x}$ of variables, a *condition* on $\bar{x}$ is a quantifier-free FO formula over $\mathcal{DB} \cup \mathcal{C} \cup \{=\}$ whose variables are included in $\bar{x}$. The special constant `null` can be used in equalities with ID variables. For each atom $R(x, y_1, \ldots, y_m, z_1, \ldots, z_n)$ of relation $R(\mathit{ID}, A_1, \ldots, A_m, F_1, \ldots, F_n) \in \mathcal{DB}$, $\{x, z_1, \ldots, z_n\} \subseteq \mathit{VAR}_{id}$ and $\{y_1, \ldots, y_m\} \subseteq \mathit{VAR}_{val}$. Atoms over $\mathcal{C}$ use only numeric variables. If α is a condition on $\bar{x}$, D is an instance of $\mathcal{DB}$ and ν a valuation of $\bar{x}$, we denote by $D \cup \mathcal{C} \models \alpha(\nu)$ the fact that $D \cup \mathcal{C}$ satisfies α with valuation ν, with standard semantics. For an atom $R(\bar{y})$ in α where $R \in \mathcal{DB}$ and $\bar{y} \subseteq \bar{x}$, if $\nu(y) = \texttt{null}$ for any $y \in \bar{y}$, then $R(\bar{y})$ is false. As will become apparent, although conditions used in HAS are quantifier-free, $\exists$FO conditions can be simulated by adding variables to $\bar{x}^T$, so we use them as shorthand whenever convenient.

Example 11. *The following $\exists$FO formula indicates that the customer in the travel booking process has chosen a pair of compatible flight and hotel and paid the discounted amount:*

$$\exists q \exists p_1 \exists p_2 \texttt{FLIGHTS}(\texttt{flight\_id}, q, \texttt{hotel\_id}) \wedge \texttt{HOTELS}(\texttt{hotel\_id}, p_1, p_2) \wedge \texttt{amount\_paid} = q + p_2.$$

We next define services of tasks. We start with internal services, which update the artifact variables and artifact relation of the task.

Definition 12. *Let $T = \langle \bar{x}^T, S^T, \bar{s}^T, \bar{x}^T_{\mathsf{in}}, \bar{x}^T_{\mathsf{out}} \rangle$ be a task schema of an artifact schema $\mathcal{A}$. An* ***internal service*** *σ of T is a tuple $\langle \pi, \psi, \delta \rangle$ where:*

- *π and ψ, called* pre-condition *and* post-condition*, respectively, are conditions over $\bar{x}^T$*
- *$\delta \subseteq \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$ is a set of* artifact relation updates*; $+S^T(\bar{s}^T)$ and $-S^T(\bar{s}^T)$ are called an* ***insertion*** *and* ***retrieval*** *of $\bar{s}^T$, respectively.*

Intuitively, an internal service of T can be called only when the current instance satisfies the pre-condition. The update of variables $\bar{x}^T$ is valid if the next instance satisfies the post-condition. Variables can be changed arbitrarily during a service activation, as long as the post condition holds. This feature allows services to also model actions by external actors who provide input into the workflow by setting the value of non-propagated variables. Such actors may even include humans or other parties whose behavior is not deterministic. For example, a bank manager carrying out a "loan decision" action can be modeled by a service whose result is stored in a variable and whose value is restricted by the post-condition to either "Approve" or "Deny". Note that deterministic actors can be modeled by simply using tighter post-conditions.

As will be seen in the formal definition, $+S^T(\bar{s}^T)$ causes an insertion of the *current* value of $\bar{s}^T$ into S^T, while $-S^T(\bar{s}^T)$ causes the removal of some non-deterministically chosen tuple of S^T and its assignment as the *next* value of $\bar{s}^T$. In particular, if $\delta = \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$, the tuple inserted by $+S^T(\bar{s}^T)$ and the one retrieved by $-S^T(\bar{s}^T)$ are generally distinct, but may be the same as a degenerate case.

Example 13. *The **ManageTrips** task has 3 internal services:* Initialize, StoreTrip *and* RetrieveTrip. Initialize *creates a new trip with* $\texttt{flight\_id} = \texttt{hotel\_id} = \texttt{null}$. *When* RetrieveTrip *is called, a previously stored trip is chosen non-deterministically and removed from* $\texttt{TRIPS}$ *for processing, and* $(\texttt{flight\_id}, \texttt{hotel\_id})$ *is set to be the chosen tuple. When* StoreTrip *is called, the current tuple* $(\texttt{flight\_id}, \texttt{hotel\_id})$ *is inserted into* $\texttt{TRIPS}$. *The latter two services are specified as follows.*

RetrieveTrip:
Pre: $\texttt{flight\_id} = \texttt{null} \wedge \texttt{hotel\_id} = \texttt{null}$
Post: $\texttt{status} =$ "Shopping"
Update: $\{-\texttt{TRIPS}(\texttt{flight\_id}, \texttt{hotel\_id})\}$

StoreTrip:
Pre: $\texttt{flight\_id} \neq \texttt{null} \vee \texttt{hotel\_id} \neq \texttt{null}$
Post: $\texttt{flight\_id} = \texttt{null} \wedge \texttt{hotel\_id} = \texttt{null} \wedge$ $\texttt{status} =$ "Shopping"
Update: $\{+\texttt{TRIPS}(\texttt{flight\_id}, \texttt{hotel\_id})\}$

Figure 3.3. Examples of two services.

As seen above, internal services of a task cause transitions on the data local to the task. Interactions among tasks are specified using two kinds of special services called the *opening*-services and *closing*-services. Specifically, each task T is equipped with an opening service σ_T^o and a closing service σ_T^c. Each non-root task T can be activated by its parent task via a call to σ_T^o which includes passing parameters to T that initialize its input variables $\bar{x}_{\text{in}}^T$. When T terminates (if ever), it returns to the parent the contents of its output variables $\bar{x}_{\text{out}}^T$ via a call to σ^c_T. Moreover, calls to σ^o_T are guarded by a condition on the parent's artifact variables, and closing calls to σ^c_T are guarded by a condition on the artifact variables of T.

Definition 14. *Let T_c be a child of a task T in $\mathcal{A}$.*
*(i) The **opening-service** $\sigma^o_{T_c}$ of T_c is a tuple $\langle \pi, f_{in} \rangle$, where π is a (pre-)condition over $\bar{x}^T$, and f_{in} is a 1-1 mapping from $\bar{x}^{T_c}_{\mathsf{in}}$ to $\bar{x}^T$ (called the input variable mapping). We denote $range(f_{in})$ by $\bar{x}^T_{T_c^{\downarrow}}$ (the variables of T passed as input to T_c).*
*(ii) The **closing-service** $\sigma^c_{T_c}$ of T_c is a tuple $\langle \pi, f_{out} \rangle$, where π is a (pre-)condition over $\bar{x}^{T_c}$, and f_{out} is a 1-1 mapping from $\bar{x}^{T_c}_{\mathsf{out}}$ to $\bar{x}^T$ (called the output variable mapping). We denote $range(f_{out})$ by $\bar{x}^T_{T_c^{\uparrow}}$, referred to as the **returned variables** from T_c. It is required that $\bar{x}^T_{T_c^{\uparrow}} \cap \bar{x}^T_{\mathsf{in}} = \emptyset$.*

Requiring $\bar{x}^T_{T_c^{\uparrow}} \cap \bar{x}^T_{\mathsf{in}} = \emptyset$ means that a returning child task cannot overwrite the input variables of the parent task, so that the values of the input variables stay unchanged throughout an execution of the task. While the definition allows the return of numeric variables, it turns out that for the purpose of verification one can assume that only ID variables are returned. One can additionally assume that the sets of variables returned by different subtasks are disjoint. The discussion of the simplifications is postponed to Section 3.3, since they must be considered in the context of the property language HLTL-FO.

For uniformity of notation, we also equip the root task T_1 with a service $\sigma^o_{T_1}$ with pre-condition *true* that initiates the computation by providing a valuation to a designated subset $\bar{x}^{T_1}_{in}$ of $\bar{x}^{T_1}$ (the input variables of T_1), and a service $\sigma^c_{T_1}$ whose pre-condition is *false* (so it never occurs in a run). For a task T we denote by Σ_T the set of its internal services, $\Sigma^{oc}_T = \Sigma_T \cup \{\sigma^o_T, \sigma^c_T\}$, $\Sigma^{obs}_T = \Sigma^{oc}_T \cup \{\sigma^o_{T_c}, \sigma^c_{T_c} \mid T_c \in child(T)\}$, and $\Sigma^{\delta}_T = \Sigma_T \cup \{\sigma^o_T\} \cup \{\sigma^c_{T_c} \mid T_c \in child(T)\}$. Intuitively, Σ^{obs}_T consists of the services observable in runs of task T and Σ^{δ}_T consists of services whose application can modify the variables $\bar{x}^T$.

Example 15. *The opening-service $\sigma^o_{T_4}$ of the **BookTrip** task has pre-condition* $\texttt{flight\_id} \neq \texttt{null} \wedge \texttt{hotel\_id} \neq \texttt{null} \wedge \texttt{status} =$ *"Shopping", meaning that both the hotel and flight* *have been chosen by the customer but are not yet paid. The input variables $\bar{x}_{in}^{T_4}$ of **BookTrip** are* $\{\texttt{flight\_id}, \texttt{hotel\_id}\}$*, which are mapped by the mapping f_{in} to variables* $\{\texttt{flight\_id}, \texttt{hotel\_id}\}$ *of **ManageTrips**. The closing-service $\sigma_{T_4}^{c}$ of the **BookTrip** task has pre-condition* $\texttt{status} =$ *"Paid", meaning that the trip was successfully paid. The output variables $\bar{x}_{out}^{T_4}$ of **BookTrip** are* $\{\texttt{status}, \texttt{amount\_paid}\}$*, which are mapped by f_{out} to the identically named variables* $\{\texttt{status}, \texttt{amount\_paid}\}$ *of **ManageTrips** (the returned variables $\bar{x}_{T_4\uparrow}^{T_1}$ from **BookTrip**).*

Definition 16. *A* Hierarchical Artifact System *(HAS) is a triple $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$, where $\mathcal{A}$ is an artifact schema, Σ is a set of services of tasks in $\mathcal{A}$ including σ_T^o and σ_T^c for each task T of $\mathcal{A}$, and Π is a condition over $\bar{x}_{in}^{T_1}$ (where T_1 is the root task).*

3.2.2 Semantics of HAS

We next define the semantics of HAS. Intuitively, a run of a HAS on a database D consists of an infinite sequence of transitions among HAS instances (also referred to as configurations, or snapshots), starting from an initial artifact tuple satisfying pre-condition Π, and empty artifact relations. At each snapshot, each active task T can open a subtask T_c if the pre-condition of the opening service of T_c holds, and the values of a subset of $\bar{x}^T$ are passed to T_c as its input variables. T_c can be closed if the pre-condition of its closing service is satisfied. When T_c is closed, the values of the return variables of T_c are sent to T. An internal service of T can only be applied after all active subtasks of T have returned their answer.

Tree of Local Runs

Because of the hierarchical structure, and the locality of task specifications, the actions of concurrently active children of a given task are independent of each other and can be arbitrarily interleaved. To capture just the essential information, factoring out the arbitrary interleavings, we first define the notion of *local run* and *tree of local runs*. Intuitively, a local run of a task consists of a sequence of services of the task, together with the transitions they cause on the task's local artifact variables and relation. The tasks' input and output are also specified. A tree of local runs captures the relationship between the local runs of tasks and those of their subtasks, including the passing of inputs and results. Then the runs of the full artifact system simply consist of all legal interleavings of transitions represented in the tree of local runs, lifted to full HAS instances (we refer to these as *global runs*). We begin by defining instances of tasks and local transitions. For a mapping M, we denote by $M[a \mapsto b]$ the mapping that sends a to b and agrees with M everywhere else.

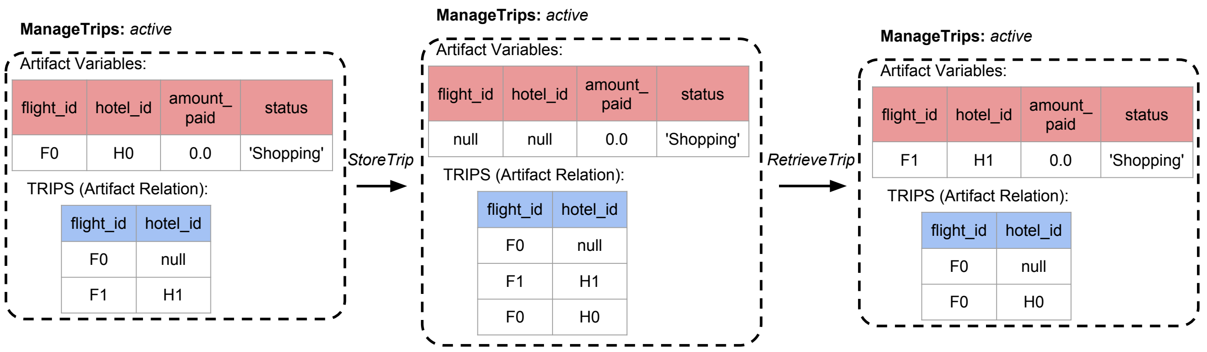
Definition 17. *Let $T = \langle \bar{x}^T, S^T, \bar{s}^T, \bar{x}^T_{\mathsf{in}}, \bar{x}^T_{\mathsf{out}} \rangle$ be a task in Γ and D a database instance over $\mathcal{DB}$. An* instance *of T is a pair (ν, S) where ν is a valuation of $\bar{x}^T$ and S an instance of S^T. For instances $I = (\nu, S)$ and $I' = (\nu', S')$ of T and a service $\sigma \in \Sigma^{obs}_T$, there is a local transition $I \xrightarrow{\sigma} I'$ if the following holds. If σ is an internal service $\langle \pi, \psi \rangle$, then:*

- *$D \cup \mathcal{C} \models \pi(\nu)$ and $D \cup \mathcal{C} \models \psi(\nu')$*
- *$\nu'(y) = \nu(y)$ for each y in $\bar{x}^T_{\mathsf{in}}$*
- *if $\delta = \{+S^T(\bar{s}^T)\}$, then $S' = S \cup \{\nu(\bar{s}^T)\}$,* [3]
- *if $\delta = \{-S^T(\bar{s}^T)\}$, then $\nu'(\bar{s}^T) \in S$ and $S' = S - \{\nu'(\bar{s}^T)\}$,*
- *if $\delta = \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$, then $\nu'(\bar{s}^T) \in S \cup \{\nu(\bar{s}^T)\}$ and $S' = (S \cup \{\nu(\bar{s}^T)\}) - \{\nu'(\bar{s}^T)\}$,*
- *if $\delta = \emptyset$ then $S' = S$.*

If $\sigma = \sigma^o_{T_c} = \langle \pi, f_{in} \rangle$ is the opening-service for a child T_c of T then $D \cup \mathcal{C} \models \pi(\nu)$, $\nu' = \nu$ and $S' = S$. If $\sigma = \sigma^c_{T_c}$ then $S = S'$, $\nu'|(\bar{x}^T - \bar{x}^T_{T_c\uparrow}) = \nu|(\bar{x}^T - \bar{x}^T_{T_c\uparrow})$ and $\nu'(z) = \nu(z)$ for every $z \in \bar{x}^T_{T_c\uparrow} \cap \mathrm{VAR}_{id}$ for which $\nu(z) \neq \texttt{null}$. Finally, if $\sigma = \sigma^c_T$ then $I' = I$.

Example 18. *Figure 5.5 shows two local transitions obtained by calling internal services* StoreTrip *and* RetrieveTrip *of **ManageTrips**. Figure 3.5 illustrates a transition caused by closing a sub-task.*

[3]All artifact relation operations preserve the order of variables/attributes.



ManageTrips: *active*

Artifact Variables:

flight_id	hotel_id	amount_ paid	status
F0	H0	0.0	'Shopping'

TRIPS (Artifact Relation):

flight_id	hotel_id
F0	null
F1	H1

ManageTrips: *active*

Artifact Variables:

flight_id	hotel_id	amount_ paid	status
null	null	0.0	'Shopping'

TRIPS (Artifact Relation):

flight_id	hotel_id
F0	null
F1	H1
F0	H0

ManageTrips: *active*

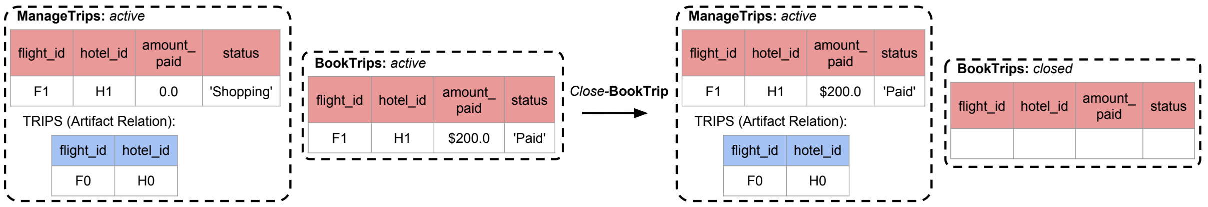
Artifact Variables:

flight_id	hotel_id	amount_ paid	status
F1	H1	0.0	'Shopping'

TRIPS (Artifact Relation):

flight_id	hotel_id
F0	null
F0	H0

Figure 3.4. Two transitions caused by the *StoreTrip* and the *RetrieveTrip* services.



ManageTrips: *active*

flight_id	hotel_id	amount_ paid	status
F1	H1	0.0	'Shopping'

TRIPS (Artifact Relation):

flight_id	hotel_id
F0	H0

BookTrips: *active*

flight_id	hotel_id	amount_ paid	status
F1	H1	$200.0	'Paid'

ManageTrips: *active*

flight_id	hotel_id	amount_ paid	status
F1	H1	$200.0	'Paid'

TRIPS (Artifact Relation):

flight_id	hotel_id
F0	H0

BookTrips: *closed*

flight_id	hotel_id	amount_ paid	status

Figure 3.5. Transition caused by a closing service.

Remark 19. *Recall that tuples retrieved from artifact relations are selected non-deterministically. For example, the* RetrieveTrip *service above extracts an arbitrary trip from the* TRIPS *relation. However, it may be useful to be able to select a particular trip for retrieval. While this capability is not explicitly provided, it can be simulated. Extracting the trip with a specified* `flight_id` *can be done as follows:*

1. *an internal service retrieves a trip non-deterministically*
2. *a subtask T is called and returns the desired* `flight_id`
3. *the retrieved trip and the chosen* `flight_id` *are passed to another subtask T' which checks whether the trip has the chosen* `flight_id`*. The run blocks (so is invalidated) if this is not the case.*

We now define local runs.

Definition 20. *Let $T = \langle \bar{x}^T, S^T, \bar{s}^T, \bar{x}^T_{\mathsf{in}}, \bar{x}^T_{\mathsf{out}} \rangle$ be a non-root task in Γ and D a database instance over $\mathcal{DB}$. A* local run *of T over D is a triple $\rho_T = (\nu_{in}, \nu_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$, where:*

- $\gamma \in \mathbb{N} \cup \{\omega\}$

- *for each $i \geq 0$, $I_i = (\nu_i, S_i)$ is an instance of T and $\sigma_i \in \Sigma_T^{obs}$*
- *ν_{in} is a valuation of $\bar{x}_{in}^T$*
- *$\sigma_0 = \sigma_T^o$ and $S_0 = \emptyset$,*
- *$\nu_0|\bar{x}_{in}^T = \nu_{in}$, $\nu_0(z) = \texttt{null}$ for $z \in \text{VAR}_{id} - \bar{x}_{\mathsf{in}}^T$ and $\nu_0(z) = 0$ for $z \in \text{VAR}_{\mathsf{val}} - \bar{x}_{\mathsf{in}}^T$*
- *if for some i, $\sigma_i = \sigma_T^c$ then $\gamma \in \mathbb{N}$ and $i = \gamma - 1$ (and ρ_T is called a* returning *local run)*
- *$\nu_{out} = \nu_{\gamma-1}|\bar{x}_{\mathsf{out}}^T$ if ρ_T is a returning run and $\perp$ otherwise*
- *a* segment *of ρ_T is a subsequence $\{(I_i, \sigma_i)\}_{i \in J}$, where J is a maximal interval $[a, b] \subseteq \{i \mid 0 \leq i < \gamma\}$ such that no σ_j is an internal service of T for $j \in [a+1, b]$. A segment J is* terminal *if $\gamma \in \mathbb{N}$ and $b = \gamma - 1$ (and is called returning if $\sigma_{\gamma-1} = \sigma_T^c$ and blocking otherwise). Segments of ρ_T must satisfy the following properties. For each child T_c of T there is at most one $i \in J$ such that $\sigma_i = \sigma_{T_c}^o$. If J is not blocking and such an i exists, there is exactly one $j \in J$ for which $\sigma_j = \sigma_{T_c}^c$, and $j > i$. If J is blocking, there is* at most *one such j.*
- *for every $0 < i < \gamma$, $I_{i-1} \xrightarrow{\sigma_i} I_i$.*

Local runs of the root task T_1 are defined as above, except that ν_{in} is a valuation of $\bar{x}_{in}^{T_1}$ such that $D \cup \mathcal{C} \models \Pi$, and $\nu_{out} = \perp$ (the root task never returns).

For a local run as above, we denote $\gamma(\rho_T) = \gamma$. Note that by definition of segment, a task can call each of its children tasks at most once between two consecutive services in Σ_T^{oc} and all of the called children tasks must complete within the segment, unless it is blocking. These restrictions are essential for decidability and are discussed in Section 3.4.

Observe that local runs take arbitrary inputs and allow for arbitrary return values from its children tasks. The valid interactions between the local runs of a tasks and those of its children is captured by the notion of *tree of local runs*.

Definition 21. *A* tree of local runs *is a directed labeled tree **Tree** where each node is an occurrence of a local run ρ_T for some task T and every edge connects a local run of a task T with a local run of a child task T_c and is labeled with a non-negative integer i (denoted $i(\rho_{T_c})$). In addition, the following properties are satisfied. Let $\rho_T = (\nu_{in}^T, \nu_{out}^T, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$ be a node* *of **Tree**, where $I_i = (\nu_i, S_i)$, $i \geq 0$. Let i be such that $\sigma_i = \sigma^o_{T_c}$ the opening service of some child T_c of T. There exists a unique edge labeled i from ρ_T to a node $\rho_{T_c} = (\nu_{in}, \nu_{out}, \{(I'_i, \sigma'_i)\}_{0 \leq i < \gamma'})$ of **Tree**, and the following hold:*

- *$\nu_i(f_{in}(z)) = \nu_{in}(z)$ for every $z \in \bar{x}^{T_c}_{\text{in}}$ where f_{in} is the input variable mapping of $\sigma^o_{T_c}$*
- *ρ_{T_c} is a returning run iff there exists $j > i$ such that $\sigma_j = \sigma^c_{T_c}$; let k be the minimum such j. Then for every $z \in \bar{x}^{T_c}_{\text{out}}$ if either (1) $\nu_{k-1}(f_{out}(z)) = \texttt{null}$[4] or (2) variable z is numeric, then $\nu_k(f_{out}(z)) = \nu_{out}(z)$, where f_{out} is the output variable mapping of $\sigma^c_{T_c}$.*

*Finally, for every node ρ_T of **Tree**, if ρ_T is blocking then there exists a child of ρ_T that is not returning (so is infinite or blocking).*

The above definition is illustrated in Fig. 3.6. Note that a tree of local runs may generally be rooted at a local run of any task of Γ. We say that ***Tree*** is *full* if it is rooted at a local run of T_1.

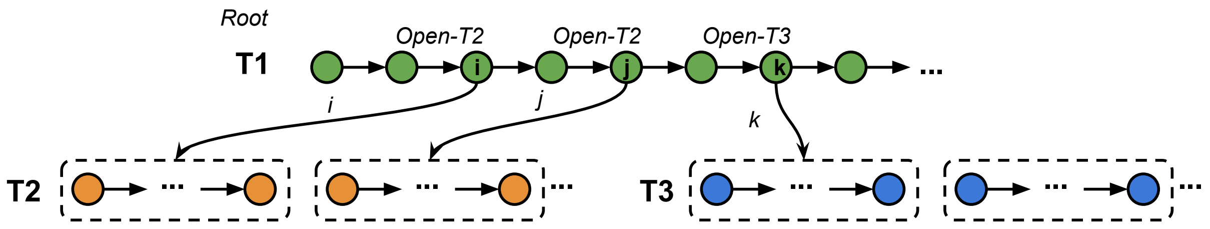


Figure 3.6. A tree of local runs.

Global runs

Intuitively, a global run of Γ on database instance D over $\mathcal{DB}$ is an infinite sequence $\rho = \{(I_i, \sigma_i)\}_{i \geq 0}$, where each I_i is an instance (ν_i, stg_i, D, S_i) of $\mathcal{A}$ and $\sigma_i \in \Sigma$, resulting from a tree of local runs by interleaving its transitions, lifted to full HAS instances. Let D be a database and ***Tree*** a full tree of local runs over D. For a local run $\rho = (\nu_{in}, \nu_{out}, \{(I_m, \sigma_m)\}_{m<\gamma})$ (where $I_m = (\nu_m, S_m)$) and $i < \gamma$, we denote by $\sigma(\rho, i) = \sigma_i$, $\nu(\rho, i) = \nu_i$, and $S(\rho, i) = S_i$. Let $\preceq$ be the pre-order on the set $\{(\rho, i) \mid \rho \in \textit{\textbf{Tree}}, 0 \leq i < \gamma(\rho)\}$ defined as the smallest reflexive-transitive relation containing the following:

[4]Although an ID variable with non-null values cannot be overwritten by a returning child task, it can be reset to `null` later by an internal transition.

1. for each node ρ and $0 \leq i \leq j < \gamma(\rho)$, $(\rho, i) \preceq (\rho, j)$
2. for each edge in ***Tree*** from ρ_T to ρ_{T_c} labeled i, $(\rho_T, i) \preceq (\rho_{T_c}, 0)$ and $(\rho_{T_c}, 0) \preceq (\rho_T, i)$. Additionally, if ρ_{T_c} is returning and m is the smallest $j > i$ for which $\sigma(\rho_T, j) = \sigma^c_{T_c}$, then $(\rho_{T_c}, \gamma(\rho_{T_c})) \preceq (\rho_T, m)$ and $(\rho_T, m) \preceq (\rho_{T_c}, \gamma(\rho_{T_c}))$.

Let $\approx$ be the equivalence relation induced by $\preceq$ (i.e., $a \approx b$ iff $a \preceq b$ and $b \preceq a$). Note that all classes of $\approx$ are singletons except for the ones induced by (2), which are of the form $\{(\rho_1, i), (\rho_2, j)\}$ where $\sigma(\rho_1, i) = \sigma(\rho_2, j) \in \{\sigma^o_T, \sigma^c_T\}$ for some task T. For an equivalence class ε of $\approx$ we denote by $\sigma(\varepsilon)$ the unique service of elements in ε. A *linearization* of $\preceq$ is an enumeration of the equivalence classes of $\approx$ consistent with $\preceq$. Consider a linearization $\{\varepsilon_i\}_{i\geq 0}$ of $\preceq$. Note that $\varepsilon_0 = (\rho_{T_1}, 0)$ and let $\nu(\rho_{T_1}, 0) = \nu_0$. A global run induced by $\{\varepsilon_i\}_{i\geq 0}$ is a sequence $\rho = \{(\bar{I}_i, \sigma_i)\}_{i\geq 0}$ such that $\sigma_i = \sigma(\varepsilon_i)$ and each $\bar{I}_i$ is an instance $(\bar{\nu}_i, stg_i, D, \bar{S}_i)$ of $\mathcal{A}$, defined inductively as follows. For $i = 0$,

- $\bar{\nu}_0(\bar{x}^{T_1}) = \nu_0(\bar{x}^{T_1})$ (and arbitrary on other variables)
- $stg_0 = \{T_1 \mapsto \texttt{active}, T_i \mapsto \texttt{init} \mid 2 \leq i \leq k\}$
- $\bar{S}_0 = \{S^{T_i} \mapsto \emptyset \mid 1 \leq i \leq k\}$.

For $i > 0$, $\bar{I}_i$ is defined as follows. Suppose first that $\varepsilon_i = \{(\rho, j)\}$ where ρ is a local run of task T and $\sigma(\rho, j)$ is an internal service of T. Then $\bar{\nu}_i = \bar{\nu}_{i-1}[\bar{x}^T \mapsto \nu(\rho, j)(\bar{x}^T)]$, $\bar{S}_i = \bar{S}_{i-1}[S^T \mapsto S(\rho, j)]$, and $stg_i = stg_{i-1}[\bar{T} \mapsto \texttt{init} \mid \bar{T} \in \mathit{desc}(T)]$. Now suppose $\varepsilon = \{(\rho_T, j), (\rho_{T_c}, 0)\}$, where T_c is a child of T, ρ_T and ρ_{T_c} are local runs of T and T_c, and $\sigma(\varepsilon) = \sigma^o_{T_c}$. Then $\bar{\nu}_i = \bar{\nu}_{i-1}[\bar{x}^{T_c} \mapsto \nu(\rho_{T_c}, 0)(\bar{x}^{T_c})]$, $\bar{S}_i = \bar{S}_{i-1}[S^{T_c} \mapsto \emptyset]$, and $stg_i = stg_{i-1}[T_c \mapsto \texttt{active}]$. Finally, suppose $\varepsilon = \{(\rho_T, j), (\rho_{T_c}, \gamma - 1)\}$ where $\sigma(\varepsilon) = \sigma^c_{T_c}$. Then $\bar{\nu}_i = \bar{\nu}_{i-1}[\bar{x}^T \mapsto \nu(\rho_T, j)(\bar{x}^T)]$, $stg_i = stg_{i-1}[T_c \mapsto \texttt{inactive}]$, and $\bar{S}_i = S_{i-1}[S^{T_c} \mapsto \emptyset]$.

We denote by $\mathcal{L}(\textbf{\textit{Tree}})$ the set of global runs induced by linearizations of $\preceq$. The set of global runs of Γ on a database D is $\mathit{Runs}_D(\Gamma) = \bigcup\{\mathcal{L}(\textbf{\textit{Tree}}) \mid \textbf{\textit{Tree}}$ is a full tree of local runs of Γ on $D\}$ and the set of global runs of Γ is $\mathit{Runs}(\Gamma) = \bigcup_D \mathit{Runs}_D(\Gamma)$.

3.3 Hierarchical LTL-FO

In order to specify temporal properties of HAS's we use an extension of LTL (linear-time temporal logic). Recall that LTL is propositional logic augmented with temporal operators $\mathbf{X}$ (next), **U** (until), **G** (always) and **F** (eventually) (e.g., see [58]). We provide a formal review of LTL next.

3.3.1 Review of LTL and LTL-FO

Here, we review the classical definition of LTL over a set P of propositions. LTL specifies properties of infinite words (ω-words) $\{\tau_i\}_{i\geq 0}$ over the alphabet consisting of truth assignments to P. Let $\tau_{\geq j}$ denote $\{\tau_i\}_{i\geq j}$, for $j \geq 0$.

The meaning of the temporal operators **X**, **U** is the following (where $\models$ denotes satisfaction and $j \geq 0$):

- $\tau_{\geq j} \models \mathbf{X}\varphi$ iff $\tau_{\geq j+1} \models \varphi$,
- $\tau_{\geq j} \models \varphi \ \mathbf{U} \ \psi$ iff $\exists k \geq j$ such that $\tau_{\geq k} \models \psi$ and $\tau_{\geq l} \models \varphi$ for $j \leq l < k$.

Observe that the above temporal operators can simulate all commonly used operators, including **G** (always) and **F** (eventually). Indeed, $\mathbf{F}\varphi \equiv$ *true* $\mathbf{U}\ \varphi$ and $\mathbf{G}\varphi \ \equiv \ \neg(\mathbf{F}\neg\varphi)$.

The standard construction of a Büchi automaton B_φ corresponding to an LTL formula φ is given in [124, 118]. The automaton B_φ has exponentially many states and accepts precisely the set of ω-words that satisfy φ.

It is sometimes useful to apply LTL on *finite* words rather than ω-words. The finite semantics we use for temporal operators is the following [39]. Let $\{\tau_i\}_{0\leq i\leq n}$ a finite sequence of truth values of P. Similarly to the above, let $\tau_{\geq j}$ denote $\{\tau_i\}_{j\leq i\leq n}$, for $0 \leq j \leq n$. The semantics of **X** and **U** are defined as follows:

- $\tau_{\geq j} \models \mathbf{X}\varphi$ iff $n > j$ and $\tau_{\geq j+1} \models \varphi$,
- $\tau_{\geq j} \models \varphi \ \mathbf{U} \ \psi$ iff $\exists k, j \leq k \leq n$ such that $\tau_{\geq k} \models \psi$ and $\tau_{\geq l} \models \varphi$ for $j \leq l < k$.

It is easy to verify that for the B_φ obtained by the standard construction [124, 118] there is a subset Q^{fin} of its states such that B_φ viewed as a finite-state automaton with final states Q^{fin} accepts precisely the finite words that satisfy φ.

LTL-FO. An extension of LTL in which propositions are interpreted as FO sentences has previously been defined to specify properties of sequences of structures [120], and in particular of runs of artifact systems [44, 36] (see Chapter 2). The extension is denoted by LTL-FO. In this thesis, we use multiple variants of LTL-FO to specify properties of different models. The differences are explained when each variant is introduced: in this chapter, Chapter 4, and Chapter 5.

3.3.2 Formal Definition of HLTL-FO

In order to specify properties of HAS's, we shall use a variant of LTL-FO, called *hierarchical* LTL-FO, denoted HLTL-FO. Intuitively, an HLTL-FO formula uses as building blocks LTL-FO formulas acting on local runs of individual tasks, referring only to the database and local data, and can recursively state HLTL-FO properties on runs resulting from calls to children tasks. This closely mirrors the hierarchical execution of tasks, and is a natural fit for this computation model. In addition to its naturalness, the choice of HLTL-FO has several technical justifications. First, verification of LTL-FO (and even LTL) properties is not possible for HAS's. Specifically, let LTL(Σ) be LTL using the services Σ as its set of propositions. For a global run $\{(I_i, \sigma_i)\}_{i \geq 0}$, proposition σ holds in (I_i, σ_i) if $\sigma = \sigma_i$ (thus, formulas in LTL(Σ) express properties of the sequence of services in a global run).

Theorem 22. *It is undecidable, given an LTL(Σ) formula φ and a HAS $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$, whether every global run of Γ satisfies φ.*

The proof, provided in Appendix 3.8.1, is by reduction from repeated state reachability in VASS with resets and bounded lossiness, whose undecidability follows from [96]. Essentially, when defined on global runs, LTL is expressive enough to encode the transitions of a VASS using the interleavings of multiple tasks. When combined with resets of counters, which can be simulated by opening/closing of tasks, verification becomes undecidable.

Another technical argument in favor of HLTL-FO is that it only expresses properties that are invariant under interleavings of independent tasks. Interleaving invariance is not only a natural soundness condition, but also allows more efficient model checking by *partial-order reduction* [104]. Moreover, HLTL-FO enjoys a pleasing completeness property: it expresses, in a reasonable sense, *all* interleaving-invariant LTL-FO properties of HAS's. This is discussed at the end of the section.

To illustrate the difference between LTL-FO and HLTL-FO, we exhibit a simple LTL property that is not expressible in HLTL.

Example 23. *Referring to our travel booking example, suppose that the opening services of* ***AddFlight*** *and* ***AddHotel*** *have preconditions* `flight_id` $=$ `null` *and* `hotel_id` $=$ `null`*, respectively. Thus, the two tasks may be active at the same time. Suppose that* ***AddFlight*** *has an internal service* ChooseFlight *and* ***AddHotel*** *has an internal service* ChooseHotel*. Consider the LTL property*

$$\mathbf{G}\left((\sigma^o_{\textbf{\textit{AddFlight}}} \wedge \mathbf{F}\ \sigma^o_{\textbf{\textit{AddHotel}}}) \rightarrow (\ \neg\text{ChooseHotel}\ \mathbf{U}\ \text{ChooseFlight}\ )\right)$$

stating that whenever the ***AddFlight*** *task is called before the* ***AddHotel*** *task, the hotel is not chosen before the flight. Clearly, this property is violated by the example, because it is not invariant with respect to legal interleavings of services. Indeed, when* ***AddFlight*** *and* ***AddHotel*** *are simulteneously active, their internal services may interleave arbitrarily. Such properties are conveniently filtered out by HLTL, which only expresses interleaving-invariant properties.*

We next define HLTL-FO. We first define the propositional version of the language, HLTL. Similarly to LTL-FO, HLTL-FO formulas are obtained by interpreting the propositions as statements about instances of tasks in a run.

Definition 24. *Let $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ be an artifact system where $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$. An HLTL formula φ over a task T of $\mathcal{H}$ is an expression defined as follows:*

$$\varphi ::= \Sigma_T^{obs} \mid P_T^{\varphi} \mid [\psi]_{T_c} \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi \mid \mathbf{F}\,\varphi \mid \mathbf{G}\,\varphi \mid (\varphi \wedge \varphi) \mid (\neg\varphi)$$

where P_T^{φ} is a finite set of propositions and ψ is an HLTL formula over task $T_c \in \text{child}(T)$. Additionally, $P_{T'}^{\varphi} \cap P_{T''}^{\varphi} = \emptyset$ for all distinct T', T'' in $\mathcal{H}$. The set of HLTL formulas over T is denoted HLTL(T).

Intuitively, a formula $[\psi]_{T_c}$ holds in a given configuration if T makes a call to T_c and the run of T_c resulting from the call satisfies ψ.

For an HLTL formula φ, we denote $P_{\mathcal{H}}^{\varphi} = \bigcup_{T \in \mathcal{H}} P_T^{\varphi}$. An HLTL-FO formula over task T is obtained from an HLTL formula over T by interpreting each proposition in $P_{\mathcal{H}}^{\varphi}$ as a quantifier-free FO formula referring to the variables and artifact relations of the tasks, and a fixed specified set of global variables. Informally, a proposition of P_T^{φ} mapped by f to a quantifier-free FO formula holds in a given configuration of T if the formula is true in that configuration. We next formally define HLTL-FO formulas.

Definition 25. *Let $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ be an artifact system where $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$. Let $\bar{y}$ be a finite sequence of variables in $\text{VAR}_{id} \cup \text{VAR}_{\text{val}}$ disjoint from $\bigcup_{T \in \mathcal{H}} \bar{x}^T$, called* global variables. *Let $\mathcal{C}_T$ be the set of conditions on $\bar{x}^T \cup \bar{y}$ extended by allowing atoms of the form $S^T(\bar{z})$ in which all variables in $\bar{z}$ are in $\bar{y} \cap \text{VAR}_{id}$. An HLTL-FO formula over task T using global variables $\bar{y}$ is a pair (φ, f) (denoted for conciseness φ_f) where φ is an HLTL formula over T and f is a mapping on $P_{\mathcal{H}}^{\varphi}$ such that $f(p) \in \mathcal{C}_{T'}$ for every $p \in P_{T'}^{\varphi}$. An HLTL-FO formula over Γ is an expression $\forall \bar{y} \varphi_f$, where φ_f is an HLTL-FO formula over task T_1 using global variables $\bar{y}$.*

Since HLTL-FO properties depend on local runs of tasks and their relationship to local runs of their descendants, their semantics is naturally defined using the full trees of local runs. We first define satisfaction by a local run in the tree, of HLTL-FO formulas with no global variables.

This is done recursively. Let ***Tree*** be a full tree of local runs of Γ over some database D. Let φ_f be an HLTL-FO formula for task T, with no global variables. If we associate to each expression $[\psi]_{T_c}$ in φ a distinct proposition $[\psi]_{T_c}^{prop}$, φ can be viewed as an LTL formula using propositions in $P_T^{\varphi} \cup \Sigma_T^{obs} \cup \{[\psi]_{T_c}^{prop} \mid \psi \in HLTL(T_c), T_c \in \textit{child}(T)\}$. Let $\rho_T = (\nu_{in}, \nu_{out}, \{(I_i, \sigma_i)\}_{i<\gamma})$ be a local run of T in ***Tree***. For each configuration (I_j, σ_j), we define the truth assignment induced on the propositions of φ by the function f. A proposition $\sigma \in \Sigma_T^{obs}$ holds in (I_j, σ_j) if $\sigma = \sigma_j$. For $p \in P_T^{\varphi}$, its induced truth value is that of the FO formula $f(p)$ in I_j. Finally, the induced truth value of $[\psi]_{T_c}^{prop}$ in (I_j, σ_j) is true iff $\sigma_j = \sigma_{T_c}^{o}$ and the local run of T_c connected to ρ_T in ***Tree*** by an edge labeled j satisfies the HLTL-FO formula ψ_f. The formula φ_f is satisfied if the sequence of induced truth values of its propositions via f satisfies φ. Note that ρ_T may be finite, in which case a finite variant of the LTL semantics is used [39] (see Appendix 3.3.1).

A full tree of local runs satisfies an HLTL-FO formula φ_f over T_1 if its root (a local run of T_1) satisfies φ_f. Finally, let $\varphi_f(\bar{y})$ be an HLTL-FO over T_1 with global variables $\bar{y}$. Then $\forall \bar{y} \varphi_f(\bar{y})$ is satisfied by ***Tree***, denoted ***Tree*** $\models \forall \bar{y} \varphi_f(\bar{y})$, if for every valuation ν of $\bar{y}$, ***Tree*** satisfies φ_{f^ν} where f^ν is obtained from f by replacing each y in $f(p)$ by $\nu(y)$ for every $p \in P$. Finally, Γ satisfies $\forall \bar{y} \varphi_f(\bar{y})$, denoted $\Gamma \models \forall \bar{y} \varphi_f(\bar{y})$, if ***Tree*** $\models \forall \bar{y} \varphi_f(\bar{y})$ for every database instance D and tree of local runs ***Tree*** of Γ on D.

The semantics of HLTL-FO on trees of local runs of a HAS also induces a semantics on the global runs of the HAS. Let $\forall \bar{y} \varphi_f(\bar{y})$ be an HLTL-FO formula and $\rho \in \mathcal{L}(\textbf{\textit{Tree}})$, where ***Tree*** is a full tree of local runs of Γ. We say that ρ satisfies $\forall \bar{y} \varphi_f(\bar{y})$ if ***Tree*** satisfies $\forall \bar{y} \varphi_f(\bar{y})$. This is well defined in view of the following easily shown fact: if $\rho \in \mathcal{L}(\textbf{\textit{Tree}}_1) \cap \mathcal{L}(\textbf{\textit{Tree}}_2)$ then $\textbf{\textit{Tree}}_1 = \textbf{\textit{Tree}}_2$.

Example 26. *The following property of the travel booking workflow can be specified in HLTL-FO:* if a discount is applied to the hotel reservation, then a compatible flight must be purchased without cancellation. *One typical way to defeat the policy would be for a user to first book the flight and the hotel with the discount price, but next cancel the flight trying to avoid paying a* *penalty. Detecting such bugs can be subtle, especially when they involve multiple tasks. The following HLTL-FO property of task **ManageTrips** says "if **BookTrip** is called and the discount is applied, then if **CancelTrip** is called next and the customer cancels the flight, then the hotel discount must also be canceled and deducted from the flight refund". The property is specified as the formula φ_f, where*

$$\varphi = \mathbf{G}\left(\texttt{Discounted} \rightarrow \mathbf{X}\left(\sigma^o_{\mathbf{T5:CancelTrip}} \rightarrow [\mathbf{G}(\mathit{CancelFlight} \rightarrow \texttt{Refund})]_{\mathbf{T5:CancelTrip}}\right)\right),$$

*$CancelFlight$ is the name of the service for canceling only the flight in **CancelTrip**, and f interprets the proposition* `Discounted` *as the subformula defined in Example 99, and* `Refund` *as the formula*

$$\exists q \exists p_1 \exists p_2\ \texttt{FLIGHTS}(\texttt{flight\_id}, q, \texttt{hotel\_id}) \wedge \texttt{HOTELS}(\texttt{hotel\_id}, p_1, p_2) \wedge \\ \texttt{amount\_refunded} = q - (p_1 - p_2).$$

Simplifications. Before proceeding, we note that several simplifications to HLTL-FO formulas and HAS specifications can be made without impact on verification. First, although useful at the surface syntax, the global variables, as well as set atoms, can be easily eliminated from the HLTL-FO formula to be verified (Lemma 89 in Appendix 3.8.2). It is also useful to note that one can assume, without loss of generality, two simplifications on artifact systems regarding the interaction of tasks with their subtasks: (i) for every task T, the set of variables passed to subtasks is disjoint with the set of variables returned by subtasks, and (ii) all variables returned by subtasks are non-numeric (Lemma 90 in Appendix 3.8.2). In view of the above, we henceforth consider only properties with no global variables or set atoms, and artifact systems simplified as described.

Checking HLTL-FO properties using automata. We next show how to check HLTL-FO properties of trees of local runs of artifact systems. Before we do so, recall the standard construction of a Büchi automaton B_φ corresponding to an LTL formula φ [124, 118]. The automaton B_φ has exponentially many states and accepts precisely the set of ω-words that satisfy φ. Recall that we are interested in evaluating LTL formulas φ on both infinite *and* finite runs. It is easily seen that for the B_φ obtained by the standard construction there is a subset Q^{fin} of its states such that B_φ viewed as a finite-state automaton with final states Q^{fin} accepts precisely the finite words that satisfy φ (details omitted).

Consider now an artifact system Γ and let φ_f be an HLTL-FO formula over Γ. Consider a full tree ***Tree*** of local runs. For task T, denote by Φ_T the set of sub-formulas $[\psi]_T$ occurring in φ and by 2^{Φ_T} the set of truth assignments to these formulas. For each T and $\eta \in 2^{\Phi_T}$, let $B(T, \eta)$ be the Büchi automaton constructed from the formula

$$\left(\wedge_{\psi \in \Phi_T, \eta(\psi)=1} \psi\right) \wedge \left(\wedge_{\psi \in \Phi_T, \eta(\psi)=0} \neg\psi\right)$$

and define the collection of automata $\mathcal{B}_\varphi = \{B(T, \eta) \mid T \in \mathcal{H}, \eta \in 2^{\Phi_T}\}$.

We now define acceptance of ***Tree*** by $\mathcal{B}_\varphi$. An *adornment* of ***Tree*** is a mapping α associating to each edge from ρ_T to ρ_{T_c} a truth assignment in $2^{\Phi_{T_c}}$. ***Tree*** is accepted by $\mathcal{B}_\varphi$ if there exists an adornment α such that:

- for each local run ρ_T of T with no outgoing edge and incoming edge with adornment η, ρ_T is accepted by $B(T, \eta)$
- for each local run ρ_T of T with incoming edge labeled by η, $\alpha(\rho_T)$ is accepted by $B(T, \eta)$, where $\alpha(\rho_T)$ extends ρ_T by assigning to each configuration $(\rho_j, \sigma^o_{T_c})$ the truth assignment in $2^{\Phi_{T_c}}$ adorning its outgoing edge labeled j. (Recall that in configurations (I_j, σ_j) for which $\sigma_j \neq \sigma^o_{T_c}$, all formulas in Φ_{T_c} are *false* by definition.)
- $\alpha(\rho_{T_1})$ is accepted by the Büchi automaton B_φ where $\alpha(\rho_{T_1})$ is defined as above.

The definition of acceptance is illustrated in Figure 3.7. The following can be shown.

Lemma 27. *A full tree of local runs **Tree** satisfies φ_f iff **Tree** is accepted by $\mathcal{B}_\varphi$.*

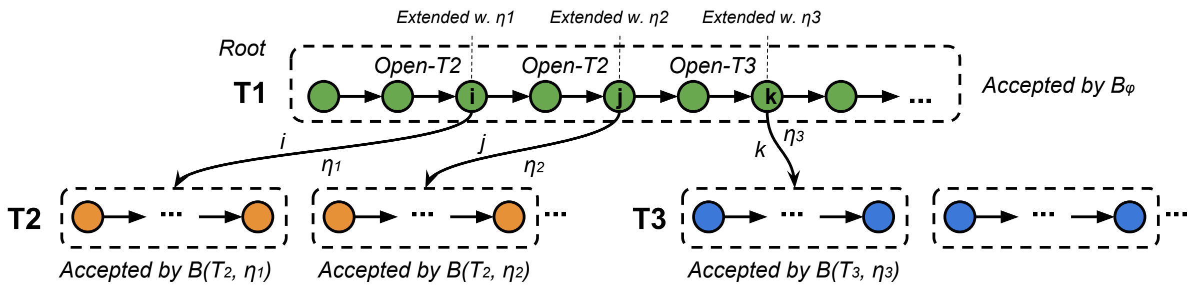


Figure 3.7. A tree of local runs accepted by $\mathcal{B}_\varphi$.

HLTL-FO vs. interleaving-invariant LTL-FO. We next show that HLTL-FO expresses, in a reasonable sense, all interleaving-invariant LTL-FO properties. We consider a notion of interleaving-invariance of LTL-FO formulas based on their propositional structure, rather than the specifics of the propositions' interpretation (which may lead to "accidental" invariance). In view of Lemma 89, we consider only formulas with no global variables or set atoms. We first recall the logic LTL-FO, slightly adapted to our context. Let $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ be a HAS where $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$. An LTL-FO formula φ_f over Γ consists of an LTL formula φ with propositions $P \cup \Sigma$ together with a mapping f associating to each $p \in P$ a condition over $\bar{x}^T$ for some $T \in \mathcal{H}$ (and we say that $f(p)$ is over T) . Satisfaction of φ_f on a global run $\rho = \{(I_i, \sigma_i)\}_{i \geq 0}$ of Γ on database D, where $I_i = (\nu_i, stg_i, D, S_i)$, is defined as usual, modulo the following:

- $f(p)$ over T holds in (I_i, σ_i) iff $stg_i(T) = \texttt{active}$ and the condition $f(p)$ on $\nu_i(\bar{x}^T)$ holds;
- proposition σ in Σ holds in (I_i, σ_i) if $\sigma = \sigma_i$.

Thus, the information about (I_i, σ_i) relevant to satisfaction of φ_f consists of σ_i, the stage of each task (active or not), and the truth values in I_i of $f(p)$ for $p \in P$.

We now make more precise the notion of (propositional) invariance under interleavings. Consider an LTL-FO formula φ_f over Γ. Invariance under interleavings is a property of the propositional formula φ (so independent on the interpretation of propositions provided by f). Let $P \cup \Sigma$ be the set of propositions of φ and let P_T denote the subset of P for which $f(p)$ is a condition over $\bar{x}^T$. Thus, $\{P_T \mid T \in \mathcal{H}\}$ is a partition of P. We define the set $\mathcal{L}(\Gamma)$ of

ω-words associated to Γ, on which φ operates. The alphabet, denoted $\mathbf{A}(\Gamma)$, consists of all triples (κ, stg, σ) where $\sigma \in \Sigma$, κ is a truth assignment to the propositions in P, and stg is a mapping associating to each $T \in \mathcal{H}$ its stage (`active`, `init` or `inactive`). An ω-word $\{(\kappa_i, stg_i, \sigma_i)\}_{i\geq 0}$ over $\mathbf{A}(\Gamma)$ is in $\mathcal{L}(\Gamma)$ if the following hold:

1. for each $i > 0$, if $\sigma_i \in \Sigma_T^\delta$, then κ_i and κ_{i-1} agree on all $P_{\bar{T}}$ where $\bar{T} \neq T$;
2. the sequence of calls, returns, and internal services obeys the conditions on service sequences in global runs of Γ;
3. for each $i > 0$ and $T \in \mathcal{H}$, $stg_i(T)$ is the stage of T as determined by the sequence of calls and returns in $\{\sigma_j\}_{j<i}$.

The formal definition of (2) and (3) mimics closely the analogous definition of global runs of HAS's (omitted). Consider an ω-word $u = \{(\kappa_i, stg_i, \sigma_i)\}_{i\geq 0}$ in $\mathcal{L}(\Gamma)$. We define the partial order $\preceq_u$ on $\{i \mid i \geq 0\}$ as the reflexive-transitive closure of the relation consisting of all pairs (i, j) such that $i < j$ and for some T, $\sigma_i, \sigma_j \in \Sigma_T^{obs}$. Observe that 0 is always the minimum element in $\preceq_u$. A linearization of $\preceq_u$ is a total order on $\{i \mid i \geq 0\}$ containing $\preceq_u$. One can represent a linearization of $\preceq_u$ as a sequence $\{i_j \mid j \geq 0\}$ such that $i_n \preceq_u i_m$ implies that $n \leq m$. For each such linearization α, we define the ω-word $u_\alpha = \{(\bar{\kappa}_j, \overline{stg}_j, \sigma_{i_j})\}_{j\geq 0}$ in $\mathcal{L}(\Gamma)$ as follows. The stage function is the one determined by the sequence of services. The functions $\bar{\kappa}_j$ are defined by induction as follows:

- $\bar{\kappa}_0 = \kappa_0$;
- if $j > 0$ and $\sigma_{i_j} \in \Sigma_T^\delta$ then $\bar{\kappa}_j = \bar{\kappa}_{j-1}[P_T \mapsto \kappa_{i_j}(P_T)]$

Intuitively, u_α is obtained from u by commuting actions that are incomparable with respect to $\preceq_u$, yielding the linearization α. We note that the relation $\preceq_u$ is the analog to our setting of Mazurkiewicz traces, used in concurrent systems to capture dependencies among process actions [97, 56, 55].

Definition 28. *An LTL-FO formula φ_f over Γ is propositionally invariant with respect to interleavings if for every $u \in \mathcal{L}(\Gamma)$ and linearization α of $\preceq_u$, $u \models \varphi$ iff $u_\alpha \models \varphi$.*

We can show the following (see Appendix 3.8.3).

Theorem 29. *HLTL-FO expresses precisely the LTL-FO properties of HAS's that are propositionally invariant with respect to interleavings.*

3.4 Restrictions and Undecidability

We briefly review the main restrictions imposed on the HAS model and motivate them by showing that they are needed to ensure decidability of verification. Specifically, recall that the following restrictions are placed:

1. in an internal transition of a given task (caused by an internal service), only the input parameters of the task are explicitly propagated from one artifact tuple to the next
2. each task may overwrite upon return only `null` variables in the parent task
3. the artifact variables of a task storing the values returned by its subtasks are disjoint from the task's input variables
4. an internal transition can take place only if all active subtasks have returned
5. each task has just one artifact relation
6. the artifact relation of a task is reset to empty every time the task closes
7. the tuple of artifact variables whose value is inserted or retrieved from a task's artifact relation is fixed
8. each subtask may be called at most once between internal transitions of its parent

These restrictions are placed in order to control the data flow and recursive computation in the system. Lifting any of them leads to undecidability of verification, as stated informally next.

Theorem 30. *For each $i, 1 \leq i \leq 8$, let HAS$^{(i)}$ be defined identically to HAS but without restriction (i) above. It is undecidable, given a HAS$^{(i)}$ Γ and an HLTL-FO formula φ_f over Γ, whether $\Gamma \models \varphi_f$.*

The proofs of undecidability for (1)-(7) are by reduction from the Post Correspondence Problem (PCP) [106, 115]. They make no use of arithmetic, so undecidability holds even without arithmetic constraints. The only undecidability result relying on arithmetic is (8). Indeed, restriction (8) can be lifted in the absence of numeric variables, with no impact on decidability or complexity of verification. This is because restriction (2) ensures that even if a subtask is called repeatedly, only a bounded number of calls have a non-vacuous effect.

The proofs using a reduction from the PCP rely on the same main idea: removal of the restriction allows to extract from the database a path of unbounded length in a labeled graph, and check that its labels spell a solution to the PCP. For illustration, the proof of undecidability for (2) using this technique is sketched in Appendix 3.9.

We claim that the above restrictions remain sufficiently permissive to capture a wide class of applications of practical interest. This is confirmed by numerous examples of practical business processes modeled as artifact systems, that we encountered in our collaboration with IBM The restrictions limit the recursion and data flow among tasks and services. In practical workflows, the required recursion is rarely powerful enough to allow unbounded propagation of data among services. Instead, as also discussed in [36], recursion is often due to two scenarios:

- allowing a certain task to undo and retry an unbounded number of times, with each retrial independent of previous ones, and depending only on a context that remains unchanged throughout the retrial phase (its input parameters). A typical example is repeatedly providing credit card information until the payment goes through, while the order details remain unchanged.
- allowing a task to batch-process an unbounded collection of records, each processed independently, with unchanged input parameters (e.g. sending invitations to an event to all attendants on the list, for the same event details).

Such recursive computation can be expressed with the above restrictions. Moreover, as discussed in Chapters 4 and 5, HAS can express a realistic benchmark of workflows obtained from existing sets of business process specifications and properties by extending them with data-aware features.

3.5 Verification Without Arithmetic

In this section we consider verification for the case when the artifact system and the HLTL-FO property have no arithmetic constraints. We show in Section 3.6 how our approach can be extended when arithmetic is present.

The roadmap to verification is the following. Let Γ be a HAS and φ_f an HLTL-FO formula over Γ. To verify that every tree of local runs of Γ satisfies φ_f, we check that there is no tree of local runs satisfying $\neg\varphi_f$, or equivalently, accepted by $\mathcal{B}_{\neg\varphi}$. Since there are infinitely many trees of local runs of Γ due to the unbounded data domain, and each tree can be infinite, an exhaustive search is impossible. We address this problem by developing a symbolic representation of trees of local runs, called *symbolic tree of runs*. The symbolic representation is subtle for several reasons. First, unlike the representations in [44, 36], it is not finite state. This is because summarizing the relevant information about artifact relations requires keeping track of the number of tuples of various isomorphism types. Second, the symbolic representation does not capture the full information about the actual runs, but just enough for verification. Specifically, we show that for every HLTL-FO formula φ_f, there exists a tree of local runs accepted by $\mathcal{B}_{\varphi}$ iff there exists a symbolic tree of runs accepted by $\mathcal{B}_{\varphi}$. We then develop an algorithm to check the latter. The algorithm relies on reductions to state reachability problems in Vector Addition Systems with States (VASS) [18].

One might wonder whether there is a simpler approach to verification of HAS, that reduces it to verification of a flat system (consisting of a single task). This could indeed be done in the absence of artifact relations, by essentially concatenating the artifact tuples of the tasks along the hierarchy that are active at any given time, and simulating all transitions by internal services. However, there is strong evidence that this is no longer possible when tasks are equipped with artifact relations. First, a naive simulation using a single artifact relation would require more powerful updating capabilities (e.g. resetting artifact relations to be empty) than available in the model. Adding these capabilities would result in a model expressive enough to simulate vector addition systems with resets where verification is undecidable [96]. Moreover, Theorem 22 shows that LTL is undecidable for hierarchical systems, whereas the results in this section imply that it is decidable for flat ones (as it coincides with HLTL for single tasks). While this does not rule out a simulation, it shows that there can be no effective simulation natural enough to be extensible to LTL properties. A reduction to the model of [36] is even less plausible, because of the lack of artifact relations. Note that, even if a reduction were possible, the results of [36] would be of no help in obtaining our lower complexities for verification, since the algorithm provided there is non-elementary in all cases.

We next embark upon the development outlined above.

3.5.1 Symbolic Representation

We begin by defining the symbolic analog of a local run, called *local symbolic run*. The symbolic tree of runs is obtained by connecting the local symbolic runs similarly to the way local runs are connected in trees of local runs.

Each local symbolic run is a sequence of symbolic representations of an actual instance within a local run of a task T. The representation has the following ingredients:

1. an equality type of the artifact variables of T and the elements in the database reachable from them by navigating foreign keys up to a specified depth $h(T)$. This is called the T*-isomorphism type* of the variables.
2. the T-isomorphism type of the input and return variables (if representing a returning local run)
3. for each T-isomorphism type of the set variables of T together with the input variables, the net number of insertions of tuples of that type in S^T.

Intuitively, (1) and (2) are needed in order to ensure that the assumptions made about the database while navigating via foreign keys in tasks and their subtasks are consistent. The depth $h(T)$ is chosen to be sufficiently large to ensure the consistency. (3) is required in order to make sure that a retrieval from S^T of a tuple with a given T-isomorphism type is allowed only when sufficiently many tuples of that type have been inserted in S^T.

We now formally define the symbolic representation, starting with T-isomorphism type. Let $\bar{x}^T$ be the variables of T. We define $h(T)$ as follows. Let FK be the foreign key graph of the schema $\mathcal{DB}$ and $F(n)$ be the maximum number of distinct paths of length at most n starting from any relation R in FK. Let $h(T) = 1 + |\bar{x}^T| \cdot F(\delta)$ where $\delta = 1$ if T is a leaf task and $\delta = \max_{T_c \in child(T)} h(T_c)$ otherwise.

Note that when the schema $\mathcal{DB}$ is acyclic, the maximum depth $h(T)$ is trivially bounded since starting from any arbitrary entry in the database, the longest path obtained by navigations with the keys/foreign keys has length bounded by the number of relations in $\mathcal{DB}$. However, this is not the case when $\mathcal{DB}$ is cyclic, as the paths can be infinite. The maximum depth $h(T)$ of navigations is now determined by the number of variables in each task and the height of the hierarchy. Intuitively, when T is a leaf task, the maximum depth is bounded by the number of variables in T because the longest navigation path is obtained when all variables are used to form the path. When T is a non-leaf task, a path can be obtained by chaining the navigation paths in multiple child tasks by passing input and return variables, which gives the above recursive definition of $h(T)$. We explain this in more detail in the proof of Lemma 49.

We next define expressions that denote navigation via foreign keys starting from the set of id variables $\bar{x}_{id}^T$ of T. For each $x \in \bar{x}_{id}^T$ and $R \in \mathcal{DB}$, let x_R be a new symbol. An expression is a sequence $\xi_1.\xi_2.\dots.\xi_m$, where $\xi_1 = x_R$ for some $x \in \bar{x}_{id}^T$ and $R \in \mathcal{DB}$, ξ_2 is an attribute of R, and for each i, $2 \leq i < m$, ξ_i is a foreign key and ξ_{i+1} is an attribute in the relation referenced by ξ_i. We define the length of $\xi_1.\xi_2.\dots.\xi_m$ as m. A *navigation set* $\mathcal{E}_T$ is a set of expressions such that:

- for each $x \in \bar{x}_{id}^T$, $\mathcal{E}_T$ contains at most one expression x_R ($R \in \mathcal{DB}$)
- $\mathcal{E}_T$ consists of all expressions $x_R.w$ where $x_R \in \mathcal{E}_T$ and the length of $x_R.w$ is at most $h(T)$.

In other words, the expressions in $\mathcal{E}_T$ denote all possible ways of navigating via foreign keys from a given subset of $\bar{x}^T_{id}$, by paths of length at most $h(T)$. In particular, note that $\mathcal{E}_T$ is closed under prefix. We can now define T-isomorphism type. Let $\mathcal{E}^+_T = \mathcal{E}_T \cup \bar{x}^T \cup \{\texttt{null}, 0\}$. The *sort* of $e \in \mathcal{E}^+_T$ is numeric if $e \in \bar{x}^T_{\mathbb{R}} \cup \{0\}$ or $e = w.a$ where a is a numeric attribute; its sort is $\texttt{null}$ if $e = \texttt{null}$ or $e = x \in \bar{x}^T_{id}$ and $x_R \notin \mathcal{E}_T$ for all $R \in \mathcal{DB}$; and its sort is ID(R) for $R \in \mathcal{DB}$ if $e = x_R$, or $e = x \in \bar{x}^T_{id}$ and $x_R \in \mathcal{E}_T$, or $e = w.f$ where f is a foreign key referencing R.

Definition 31. *A T-isomorphism type τ consists of a navigation set $\mathcal{E}_T$ together with an equivalence relation $\sim_\tau$ over $\mathcal{E}^+_T$ such that:*

- *if $e \sim_\tau e'$ then e and e' are of the same sort;*
- *for every $\{x, x_R\} \subseteq \mathcal{E}^+_T$, $x \sim_\tau x_R$;*
- *for every e of sort $\texttt{null}$, $e \sim_\tau \texttt{null}$;*
- *if $u \sim_\tau v$ and $u.f, v.f \in \mathcal{E}_T$ then $u.f \sim_\tau v.f$.*

We call an equivalence relation $\sim_\tau$ as above an *equality type* for τ. The relation $\sim_\tau$ is extended to tuples componentwise.

The intuition underlying the above definition is the following. First, the relation $\sim_\tau$ is an equivalence relation over the navigation set $\mathcal{E}_T$ extended with the variables $\bar{x}^T$ and the constants. Two expressions can be equal in $\sim_\tau$ only when they are of the same sort, meaning that they are both numeric, nulls or navigations ending with foreign key attributes referencing the ID of the same relation. Second, for an ID variable x, if an expression x_R appears in $\mathcal{E}^+_T$, this means that x contains a tuple id of relation R, and x and x_R are essentially the same. Finally, if two expressions u and v are equal, then the key and foreign key dependencies require that expressions $u.f$ and $v.f$ extending u and v with the same expression f must also be equal.

Note that τ provides enough information to evaluate conditions over $\bar{x}^T$. Satisfaction of a condition φ by an isomorphism type τ, denoted $\tau \models \varphi$, is defined as follows:

- $x = y$ holds in τ iff $x \sim_\tau y$,
- $R(x, y_1, \ldots, y_n, z_1, \ldots, z_m)$ holds in τ for relation $R(id, a_1, \ldots, a_n, f_1, \ldots, f_m)$ where the a_i's and f_i's are numeric and foreign key attributes respectively, iff $\{x_R.a_1, \ldots, x_R.a_n, x_R.f_1, \ldots,$

$x_R.f_m\} \subseteq \mathcal{E}_T$, and $(y_1, \dots, y_n, z_1, \dots, z_m) \sim_\tau (x_R.a_1, \dots, x_R.a_n, x_R.f_1, \dots, x_R.f_m)$.

- Boolean combinations of conditions are standard.

Example 32. *Figure 4.4 shows an example of a T-isomorphism type. The database schema contains two relations $R(\text{ID}, A)$ and $S(\text{ID}, B, C)$ where A is a foreign key attribute referencing* ID *of S and $\{B, C\}$ are numeric attributes. The task T contains 3 ID variables $\{x, y, z\}$. The T-isomorphism type has the following expressions: variables $\{x, y, z\}$, constants $\{0, \texttt{null}\}$ and a set of navigations $\{x_R.A, y_R.A, x_R.A.B, y_R.A.B, x_R.A.C, y_R.A.C\}$. Since the schema is acyclic, the navigation depth $h(T)$ is bounded by the depth of the foreign key graph. The edges in Fig. 4.4 represent the equality type $\sim_\tau$ where two expressions e and e' are connected if $e \sim_\tau e'$. Note that since $x_R.A \sim y_R.A$, to ensure that the FDs are satisfied, we must also have $x_R.A.B \sim_\tau y_R.A.B$ and $x_R.A.C \sim_\tau y_R.A.C$ in $\sim_\tau$.*

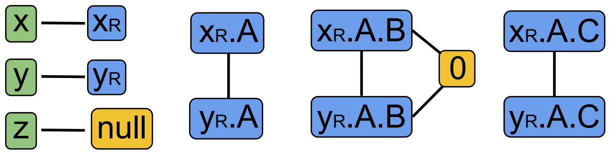


Figure 3.8. A T-isomorphism type.

Let τ be a T-isomorphism type with navigation set $\mathcal{E}_T$ and equality type $\sim_\tau$. The projection of τ onto a subset of variables $\bar{z}$ of $\bar{x}^T$ is defined as follows. Let $\mathcal{E}_T|\bar{z} = \{x_R.e \in \mathcal{E}_T | x \in \bar{z}\}$ and $\sim_\tau |\bar{z}$ be the projection of $\sim_\tau$ onto $\bar{z} \cup \mathcal{E}_T|\bar{z} \cup \{\texttt{null}, 0\}$. The projection of τ onto $\bar{z}$, denoted as $\tau|\bar{z}$, is a T-isomorphism type with navigation set $\mathcal{E}_T|\bar{z}$ and equality type $\sim_\tau |\bar{z}$. Furthermore, the projection of T-isomorphism onto $\bar{z}$ up to length k, denoted as $\tau|(\bar{z}, k)$, is defined as $\tau|\bar{z}$ with all expressions in $\mathcal{E}_T|\bar{z}$ with length more than k removed.

We apply variable renaming to isomorphism types as follows. Let f be a 1-1 partial mapping from $\bar{x}^T$ to $\mathit{VAR}_{id} \cup \mathit{VAR}_{val}$ such that $f(\bar{x}^T_{id}) \subseteq \mathit{VAR}_{id}$, $f(\bar{x}^T_{\mathbb{R}}) \subseteq \mathit{VAR}_{val}$ and $f(\bar{x}^T) \cap \bar{x}^T = \emptyset$. For a T-isomorphism type τ with navigation set $\mathcal{E}_T$, $f(\tau)$ is the isomorphism type obtained as follows. Its navigation set is obtained by replacing in $\mathcal{E}_T$ each variable x and x_R in $\mathcal{E}_T$ with $f(x)$ and $f(x)_R$, for $x \in dom(f)$. The relation $\sim_{f(\tau)}$ is the image of $\sim_\tau$ under the same substitution.

As we shall see, variable renaming and projection are applied to an isomorphism type when a subset of the variables of a task are passed as input variables to a child task. Projection is also used when a tuple is inserted in an artifact relation.

As noted earlier, a T-isomorphism type captures all information needed to evaluate a condition on $\bar{x}_T$. However, the set S^T can contain unboundedly many tuples, which cannot be represented by a finite equality type. This is handled by keeping a set of counters for projections of T-isomorphism types on the variables relevant to S^T, that is, $(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$. We refer to the projection of a T-isomorphism type onto $(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$ as a TS-isomorphism type, and denote by *TS*(T) the set of TS-isomorphism types of T. We will use counters to record the number of tuples in S^T of each TS-isomorphism type.

We can now define symbolic instances.

Definition 33. *A symbolic instance I of task T is a tuple $(\tau, \bar{c})$ where τ is a T-isomorphism type and $\bar{c}$ is a vector of integers where each dimension of $\bar{c}$ corresponds to a TS-isomorphism type.*

We denote by $\bar{c}(\hat{\tau})$ the value of the dimension of $\bar{c}$ corresponding to the TS-isomorphism type $\hat{\tau}$ and by $\bar{c}[\hat{\tau} \mapsto a]$ the vector obtained from $\bar{c}$ by replacing $\bar{c}(\hat{\tau})$ with a.

Example 34. *Examples of symbolic instances can be found in Fig. 4.5, where the task schema is the same as the one in Example 32 and the database schema is a single relation $R(\text{ID}, A)$ with a single numeric attribute A. The symbolic instances consist of the T-isomorphism types and collections of counters of TS-isomorphism types.*

Definition 35. *A* local symbolic run $\tilde{\rho}_T$ *of task T is a tuple $(\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$, where:*

- *each I_i is a symbolic instance $(\tau_i, \bar{c}_i)$ of T*
- *each σ_i is a service in Σ^{obs}_T*
- *$\gamma \in \mathbb{N} \cup \{\omega\}$ (if $\gamma = \omega$ then $\tilde{\rho}_T$ is infinite, otherwise it is finite)*
- *τ_{in}, called the input isomorphism type, is a T-isomorphism type projected to $\bar{x}^T_{\mathsf{in}}$. And $\tau_{in} \models \Pi$ if $T = T_1$.*

- *at the first instance I_0, $\tau_0|\bar{x}^T_{\mathsf{in}} = \tau_{in}$, for every $x \in \bar{x}^T_{id} - \bar{x}^T_{\mathsf{in}}$, $x \sim_{\tau_0}$ `null`, and for every $x \in \bar{x}^T_{\mathbb{R}} - \bar{x}^T_{\mathsf{in}}$, $x \sim_{\tau_0} 0$. Also $\bar{c}_0 = \bar{0}$ and $\sigma_0 = \sigma^o_T$.*
- *if for some i, $\sigma_i = \sigma^c_T$ then $\tilde{\rho}_T$ is finite and $i = \gamma - 1$ (and $\tilde{\rho}_T$ is called a* returning *run)*
- *τ_{out} is $\perp$ if $\tilde{\rho}_T$ is infinite or finite but $\sigma_{\gamma-1} \neq \sigma^c_T$, and it is $\tau_{\gamma-1}|(\bar{x}^T_{\mathsf{in}} \cup \bar{x}^T_{\mathsf{out}})$ otherwise*
- *a* segment *of $\tilde{\rho}_T$ is a subsequence $\{(I_i, \sigma_i)\}_{i \in J}$, where J is a maximal interval $[a, b] \subseteq \{i \mid 0 \leq i < \gamma\}$ such that no σ_j is an internal service of T for $j \in [a+1, b]$. A segment J is* terminal *if $\gamma \in \mathbb{N}$ and $b = \gamma - 1$. Segments of $\tilde{\rho}_T$ must satisfy the following properties. For each child T_c of T there is at most one $i \in J$ such that $\sigma_i = \sigma^o_{T_c}$. If J is not terminal and such i exists, there is exactly one $j \in J$ for which $\sigma_j = \sigma^c_{T_c}$, and $j > i$. If J is terminal, there is* at most *one such j.*
- *for every $0 < i < \gamma$, I_i is a* **successor** *of I_{i-1} under σ_i (see below).*

The successor relation is defined next. We begin with some preliminary definitions.

A TS-isomorphism type $\hat{\tau}$ is *input-bound* if for every $s \in \bar{s}^T$, $s \not\sim_{\hat{\tau}}$ `null` implies that there exists an expression $x_R.w$ in $\hat{\tau}$ such that $x \in \bar{x}^T_{\mathsf{in}}$ and $x_R.w \sim_{\hat{\tau}} s$. We denote by $TS_{ib}(T)$ the set of input-bound types in $TS(T)$. Informally, a TS-isomorphism type is input-bound if the values of all the variables in $\bar{s}^T$ are uniquely determined by the values of the input variables of T. Since the values of the input variables are fixed in a local run of T, tuples $\bar{s}^T$ reachable from them in the same run by given navigations are unique. Therefore, the counter values for these TS-isomorphism types cannot exceed 1, and are treated as special cases when updated, as shown below.

For $\hat{\tau}, \hat{\tau}' \in TS(T)$, update δ of the form $\{+S^T(\bar{s}^T)\}$ or $\{-S^T(\bar{s}^T)\}$ and mapping $\bar{c}_{ib}$ from $TS_{ib}(T)$ to $\{0, 1\}$, we define the mapping $\bar{a}(\delta, \hat{\tau}, \hat{\tau}', \bar{c}_{ib})$ from $TS(T)$ to $\{-1, 0, 1\}$ as follows. Informally, the vector $\bar{a}(\delta, \hat{\tau}, \hat{\tau}', \bar{c}_{ib})$ specifies how the current counters need to be modified to reflect the update δ. Note that $\bar{a}_0$ is the mapping sending $TS(T)$ to 0.

- if $\delta = \{+S^T(\bar{s}^T)\}$, then $\bar{a}(\delta, \hat{\tau}, \hat{\tau}', \bar{c}_{ib})$ is $\bar{a}_0[\hat{\tau} \mapsto 1]$ if $\hat{\tau}$ is not input-bound, and $\bar{a}_0[\hat{\tau} \mapsto (1 - \bar{c}_{ib}(\hat{\tau}))]$ otherwise
- if $\delta = \{-S^T(\bar{s}^T)\}$, then $\bar{a}(\delta, \hat{\tau}, \hat{\tau}', \bar{c}_{ib}) = \bar{a}_0[\hat{\tau}' \mapsto -1]$

- if δ is $\{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$ then $\bar{a}(\delta, \hat{\tau}, \hat{\tau}', \bar{c}_{ib}) = \bar{a}(\delta^+, \hat{\tau}, \hat{\tau}', \bar{c}_{ib}) + \bar{a}(\delta^-, \hat{\tau}, \hat{\tau}', \bar{c}_{ib})$ where $\delta^+ = \{+S^T(\bar{s}^T)\}$ and $\delta^- = \{-S^T(\bar{s}^T)\}$.

Next, we define the **successor** relation of symbolic instances. For symbolic instances $I = (\tau, \bar{c})$ and $I' = (\tau', \bar{c}')$, I' is a successor of I by applying service σ' iff:

- If σ' is an internal service $\langle \pi, \psi, \delta \rangle$, then for $\hat{\tau} = \tau|(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$ and $\hat{\tau}' = \tau'|(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$,
  - $\tau|\bar{x}^T_{\mathsf{in}} = \tau'|\bar{x}^T_{\mathsf{in}}$,
  - $\tau \models \pi$ and $\tau' \models \psi$,
  - $\bar{c}' \geq \bar{0}$ and $\bar{c}' = \bar{c} + \bar{a}(\delta, \hat{\tau}, \hat{\tau}', \bar{c}_{ib})$, where $\bar{c}_{ib}$ the restriction of $\bar{c}$ to $\mathit{TS}_{ib}(T)$.
- If σ' is an opening service $\langle \pi, f_{in} \rangle$ of subtask T_c, then $\tau = \tau' \models \pi$ and $\bar{c}' = \bar{c}$.
- If σ' is a closing service of subtask T_c, then for $\bar{x}^T_{const} = \bar{x}^T - \{x \in \bar{x}^T_{T_c^\uparrow} | x \sim_\tau \texttt{null}\}$, $\tau'|\bar{x}^T_{const} = \tau|\bar{x}^T_{const}$ and $\bar{c}' = \bar{c}$.
- If σ' is the closing service $\sigma^c_T = \langle \pi, f_{out} \rangle$ of T, then $\tau \models \pi$ and $(\tau, \bar{c}) = (\tau', \bar{c}')$.

Example 36. *Figure 4.5 shows an example of two symbolic transitions. There is a single database relation $R(\text{ID}, A)$, the task T has 3 variables $\{x, y, z\}$ and $\bar{s}^T = \{y, z\}$. There is no input variable. The two applied services are* "insert_yz" *and* "retrieve_yz"*:*

- *The pre-condition of* insert_yz *is $x = y$, the post-condition is $x = \texttt{null} \wedge y = \texttt{null} \wedge z = \texttt{null}$, and the set update is $\{+S^T(y, z)\}$. So when applying* insert_yz*, the current tuple (y, z) is inserted to S^T and the values of all variables are set to* null.
- *The pre-condition of* retrieve_yz *is* True*, the post-condition is $x = \texttt{null}$, and the set update is $\{-S^T(y, z)\}$. So when applying* retrieve_yz*, a tuple (y, z) will be retrieved from S^T, the variables $\{y, z\}$ are set to the retrieved tuple, and x is set to* null.

Denote by $(\tau_1, \bar{c}_1)$, $(\tau_2, \bar{c}_2)$ and $(\tau_3, \bar{c}_3)$ the 3 symbolic instances. In order for $(\tau_2, \bar{c}_2)$ to be a valid successor of $(\tau_1, \bar{c}_1)$ by applying insert_yz*, the T-isomorphism types τ_1 and τ_2 must satisfy the pre-condition and post-condition of* insert_yz *respectively, and the counter vector $\bar{c}_2$ must be obtained from $\bar{c}_1$ by incrementing the counter for the projection $\tau_1|\{y, z\}$ by 1. Similarly, in order* *to apply* retrieve_yz, *τ_2 and τ_3 must satisfy the pre-condition and post-condition of* retrieve_yz, *and the counter for the projection $\tau_3|\{y, z\}$ must be decremented by 1 in $\bar{c}_3$.*

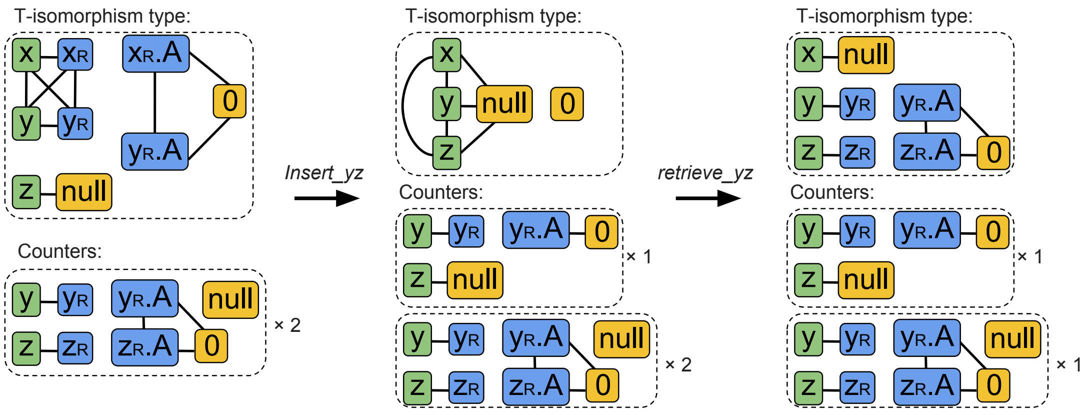


Figure 3.9. Two local symbolic transitions.

Note that there is a subtle mismatch between transitions in actual local runs and in symbolic runs. In the symbolic transitions defined above, a service inserting a tuple in S^T *always* causes the correspoding counter to increase (except for the input-bound case). However, in actual runs, an inserted tuple may collide with an already existing tuple in the set, in which case the number of tuples does *not* increase. Symbolic runs do not account for such collisions (beyond the input-bound case), which raises the danger that they might overestimate the number of available tuples and allow impossible retrievals. Fortunately, the proof of Theorem 39 shows that collisions can be ignored at no peril. More specifically, it follows from the proof that for every actual local run with collisions satisfying an HLTL-FO property there exists an actual local run without collisions that satisfies the same property. The intuition is the following. First, given an actual run with collisions, one can modify it so that only new tuples are inserted in the artifact relation, thus avoiding collisions. However, this raises a challenge, since it may require augmenting the database with new tuples. If done naively, this could result in an infinite database. The more subtle observation, detailed in the proof of Theorem 39, is that only a bounded number of new tuples must be created, thus keeping the database finite.

Definition 37. *A* symbolic tree of runs *is a directed labeled tree* **Sym** *in which each node is a local symbolic run $\tilde{\rho}_T$ for some task T, and every edge connects a local symbolic run of a task T*

*with a local symbolic run of a child task T_c and is labeled with a non-negative integer i (denoted $i(\tilde{\rho}_{T_c})$). In addition, the following properties are satisfied. Let $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$ be a node of **Sym**. Let i be such that $\sigma_i = \sigma^o_{T_c}$ for some child T_c of T. There exists a unique edge labeled i from $\tilde{\rho}_T$ to a node $\tilde{\rho}_{T_c} = (\tau'_{in}, \tau'_{out}, \{(I'_i, \sigma'_i)\}_{0 \leq i < \gamma'})$ of **Sym**, and the following hold:*

- $\tau'_{in} = f_{in}^{-1}(\tau_i)|(\bar{x}^{T_c}_{\mathsf{in}}, h(T_c))$ *where f_{in} is the input variable mapping of $\sigma^o_{T_c}$*
- $\tilde{\rho}_{T_c}$ *is a returning run iff there exists $j > i$ such that $\sigma_j = \sigma^c_{T_c}$; let k be the minimum such j. Let $\bar{x}_r = \bar{x}^T_{T_c^{\downarrow}}$ and $\bar{x}_w = \{x | x \in \bar{x}^T_{T_c^{\uparrow}}, x \sim_{\tau_{k-1}} \mathtt{null}\}$. Then $\tau_k|(\bar{x}_r \cup \bar{x}_w, h(T_c)) = ((f_{in} \circ f_{out})(\tau_{out}))|(\bar{x}_r \cup \bar{x}_w)$ where f_{out} is the output variable mapping of $\sigma^c_{T_c}$.*

For every local symbolic run $\tilde{\rho}_T$ where $\gamma \neq \omega$ and $\tau_{out} = \bot$, there exists a child of $\tilde{\rho}_T$ which is not returning.

Now consider an HLTL-FO formula φ_f over Γ. Satisfaction of φ_f by a symbolic tree of runs is defined analogously to satisfaction by local runs, keeping in mind that as previously noted, isomorphism types of symbolic instances of T provide enough information to evaluate conditions over $\bar{x}^T$. The definition of acceptance by the automaton $\mathcal{B}_\varphi$, and Lemma 27, are also immediately extended to symbolic trees of runs. We state the following.

Lemma 38. *A symbolic tree of runs **Sym** over Γ satisfies φ_f iff **Sym** is accepted by $\mathcal{B}_\varphi$.*

The key result enabling the use of symbolic trees of runs is the following.

Theorem 39. *For an artifact system Γ and HLTL-FO property φ_f, there exists a tree of local runs **Tree** accepted by $\mathcal{B}_\varphi$, iff there exists a symbolic tree of runs **Sym** accepted by $\mathcal{B}_\varphi$.*

The *only-if* part is relatively straightforward and we outline the proof in Section 3.5.2. The *if* part is non-trivial. We prove it by showing a construction of an actual database and an accepted tree of local runs from any accepted symbolic tree of runs ***Sym***. The construction has 3 major components.

- First, for each *finite* local symbolic run, we construct an actual accepted local run over a local database (Lemma 44), using a global equality type that extends the local equality types by taking into account connections across instances resulting from the propagation of input variables and insertions/retrievals of tuples from S^T, and subject to satisfaction of the key constraints. The key challenge in this step is to show that our choice of $h(T)$, the maximal navigation depth in the symbolic representations, is sufficiently large to guarantee satisfaction of all key constraints (Lemma 49).

- Next, we apply the same construction to each *infinite* local symbolic run, resulting in an accepted infinite local run over an infinite database. The infinite database is then turned into a finite one by carefully merging data values, while avoiding any inconsistencies. One of the subtleties is showing that the mismatch between symbolic and actual transitions discussed above, leading to the possible overestimation by the counters in symbolic runs of the number of tuples available in artifact relations, is not dangerous (Lemma 62).
- Finally, all finite and infinite local runs are recursively combined into a tree of local runs by renaming and merging data values stored in the variables and the local databases.

3.5.2 Only-if: from actual runs to symbolic runs

Let ***Tree*** be a tree of local runs accepted by $\mathcal{B}_\varphi$ (with database D). The construction of ***Sym*** from ***Tree*** is simple. This can be done by replacing each local run $\rho_T \in$ ***Tree*** with a local symbolic run $\tilde{\rho}_T$. More precisely, let

$$\rho_T = (\nu_{in}, \nu_{out}, \{(J_i, \sigma_i)\}_{0 \le i < \gamma})$$

be a local run in ***Tree***, where $J_i = (\nu_i, S_i)$, We construct a corresponding local symbolic run

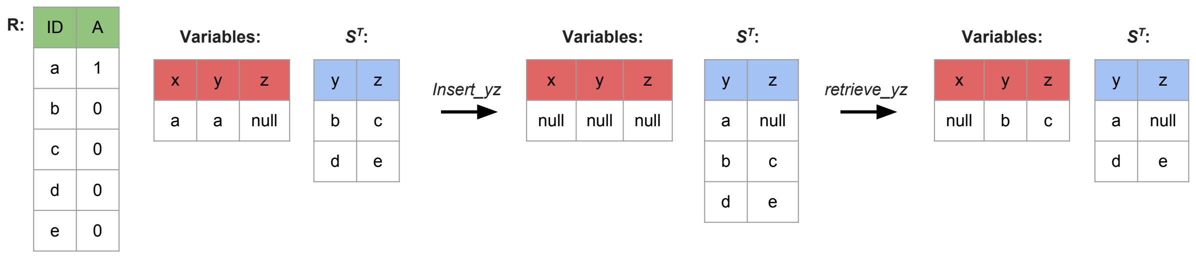
$$\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$$

For $0 \le i < \gamma$, $I_i = (\tau_i, \bar{c}_i)$ is constructed from (ν_i, S_i) as follows. The navigation set $\mathcal{E}_T$ of τ_i contains every x_R for every $x \in \bar{x}^T$ and R such that $\nu(x)$ is an ID of relation R in D. Then we define ν_i^* to be a mapping from $\mathcal{E}_T^+ = \mathcal{E}_T \cup \{0, \texttt{null}\} \cup \bar{x}^T$ to actual values, where:

- $\nu_i^*(e) = e$ if $e \in \{0, \texttt{null}\}$,
- $\nu_i^*(e) = \nu_i(x)$ for $e = x$ or $e = x_R$, and
- $\nu_i^*(e.\xi) = t.\xi$ if $\nu_i^*(e)$ is an ID of a tuple $t \in D$.

We construct the equality type $\sim_{\tau_i}$ such that for every e and e' in $\mathcal{E}_T^+$, $e \sim_{\tau_i} e'$ iff $\nu_i^*(e) = \nu_i^*(e')$. Also we let $\tau_{in} = \tau_0|\bar{x}_{\text{in}}^T$ and $\tau_{out} = \tau_{\gamma-1}|\bar{x}_{\text{in}}^T \cup \bar{x}_{\text{out}}^T$ if $\nu_{out} \neq \perp$ and $\tau_{out} = \perp$ otherwise. Since D satisfies the functional dependencies, for every τ_i and expressions e and e', $e \sim_{\tau_i} e'$ implies that $\nu_i^*(e) = \nu_i^*(e')$, so for every attribute a, if $e.a$ and $e'.a$ are in the navigation set of τ_i, then $e.a \sim_{\tau_i} e'.a$ because $\nu_i^*(e.a) = \nu_i^*(e'.a)$.

To illustrate the construction of the local symbolic runs, consider the two actual transitions shown in Figure 3.10. The above construction yields the symbolic instances shown in Figure 4.5.



ID	A
a	1
b	0
c	0
d	0
e	0

x	y	z
a	a	null

y	z
b	c
d	e

x	y	z
null	null	null

y	z
a	null
b	c
d	e

x	y	z
null	b	c

y	z
a	null
d	e

Figure 3.10. Illustration of the construction of the local symbolic runs.

By construction of the τ_i's, the following facts hold:

Fact 40. *For every condition ψ over $\bar{x}^T$, $D \models \psi(\nu_i)$ iff $\tau_i \models \psi$.*

Fact 41. *For all i, i' and $\bar{x} \subseteq \bar{x}^T$, if $\nu_i(\bar{x}) = \nu_{i'}(\bar{x})$ then $\tau_i|\bar{x} = \tau_{i'}|\bar{x}$.*

Given $\{(\tau_i, \sigma_i)\}_{0 \leq i < \gamma}$, the sequence of vectors of TS-isomorphism type counters $\{\bar{c}_i\}_{0 \leq i < \gamma}$ is uniquely defined. Let $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$. In view of Fact 40, it is easy to see that $\tilde{\rho}_T$ satisfies all items in the definition of local symbolic run that do not involve the counters. To show that $\tilde{\rho}_T$ is a local symbolic run, it remains to show that $\bar{c}_i \geq \bar{0}$ for $0 \leq i < \gamma$. To see that this holds, we associate a sequence of counter vectors $\{\tilde{c}_i\}_{0 \leq i < \gamma}$ to the local run ρ_T, where each $\tilde{c}_i$ provides, for each TS-isomorphism type $\hat{\tau}$, the number of tuples in S_i of TS-isomorphism type $\hat{\tau}$ (the TS-isomorphism type of a tuple $t \in S_i$ is defined analogously to the T-isomorphism type for each local instance). By definition, $\tilde{c}_i \geq \bar{0}$ for each $i \geq 0$. Thus it is sufficient to show that $\tilde{c}_i \leq \bar{c}_i$ for each i. We show this by induction. For $i = 0$, $\tilde{c}_0 = \bar{c}_0 = 0$. Suppose $\tilde{c}_{i-1} \leq \bar{c}_{i-1}$ and consider the transition under service σ_i in ρ_T and $\tilde{\rho}_T$. It is easily seen that $\tilde{c}_{i-1}$ and $\bar{c}_{i-1}$ are modified in the same way *except* in the case when $+S^T(\bar{s}^T) \in \delta$, $\hat{\tau}_{i-1}$ is not input-bound, and $\nu_{i-1}(\bar{s}^T) \in S_{i-1}$. In this case, if $\hat{\tau}$ is the TS-isomorphism type of $\nu_{i-1}(\bar{s}^T)$, $\tilde{c}_i(\hat{\tau}) = \tilde{c}_{i-1}(\hat{\tau})$ whereas $\bar{c}_i(\hat{\tau}) = \bar{c}_{i-1}(\hat{\tau}) + 1$. In all cases, $\tilde{c}_i \leq \bar{c}_i$. Thus, $\tilde{\rho}_T$ is a local symbolic run. The fact that ***Sym*** is a tree of symbolic local runs follows from Fact 41, which ensures the consistency of the isomorphism types passed to and from subtasks. Finally, the fact that ***Sym*** is accepted by $\mathcal{B}_\varphi$ follows from acceptance of ***Tree*** by $\mathcal{B}_\varphi$ and Fact 40.

3.5.3 If part: from symbolic runs to actual runs

We denote by **FD** the set of key dependencies in the database schema $\mathcal{DB}$ and **IND** the set of foreign key dependencies. We show the following.

Lemma 42. *For every symbolic tree of runs* **Sym** *accepted by* $\mathcal{B}_\beta$*, there exists a tree* **Tree** *of local runs accepted by* $\mathcal{B}_\beta$ *with a finite database instance* D *where* $D \models$ **FD***.*

Note that the above does not require that D satisfy **IND**. This is justified by the following.

Lemma 43. *For every tree of local runs* **Tree** *with database* $D \models$ **FD** *if* **Tree** *is accepted by* $\mathcal{B}_\beta$ *then there exists a finite database* $D' \models$ **FD** $\cup$ **IND** *such that* **Tree** *with database* D' *is also a tree of local runs accepted by* $\mathcal{B}_\beta$*.*

Proof. We can construct D' by adding tuples to D as follows. First, for each relation R such that R is empty in D, we add an arbitrary tuple t to R. Next, for each foreign key dependency $R_i[F] \subseteq R_j[ID]$, for each tuple t of R_i such that there is no tuple in R_j with id $t[F]$, we add to R_j a tuple t' where

- $t'[ID] = t[F]$, and

- $t'[attr(R_j) - \{\textit{ID}\}] = t''[attr(R_j) - \{\textit{ID}\}]$ where t'' is an existing tuple in R_j.

Tree with database D' is accepted by $\mathcal{B}_\beta$ since D' is an extension of D. Also D' is finite since the number of added tuples is at most linear in the sum of number of empty relations in D and the number of tuples in D that violate **IND**. □

To show Lemma 42, we begin with a construction of a local run ρ_T on a finite database D_T for each local symbolic run $\tilde{\rho}_T \in$ ***Sym***. The local runs are constructed so that they can be merged consistently into a tree of local runs ***Tree*** with a single finite database D. The major challenge in the construction of each ρ_T and D_T is that if $\tilde{\rho}_T$ is infinite, the size of S^T can grow infinitely, and a naive construction of ρ_T would require infinitely many distinct values in D_T. Our construction needs to ensure that D_T is always finite. For ease of exposition, we first consider the case where $\tilde{\rho}_T$ is finite and then extend the result to infinite $\tilde{\rho}_T$.

3.5.4 Handling finite local symbolic runs

Recall from the previous section that $\nu^*(e)$ denotes the value of expression e in database D_T with valuation ν of $\bar{x}^T$. By abuse of notation, we extend $\nu^*(e)$ to $e \in \{x_R.w | x \in \bar{x}^T, R \in \mathcal{DB}\} \cup \bar{x}^T \cup \{0, \texttt{null}\}$ where there is no restriction on the length of w. So for expression $e = x_R.w$, $\nu^*(e)$ is the value in D_T obtained by foreign key navigation starting from the value $\nu^*(x)$ at relation R and by the sequence of attributes w, if such a value exists. Note that ν^* may be only partially defined since D_T may not satisfy all foreign key constraints. Analogously, we define $\nu^*_{in}(e)$ to be the value of e in D_T at valuation ν_{in} and $\nu^*_{out}(e)$ to be the value of e in D_T at valuation ν_{out}.

We prove the following, showing the existence of an actual local run corresponding to a finite local symbolic run. The lemma provides some additional information used when merging local runs into a final tree of runs.

Lemma 44. *For every finite local symbolic run $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$ ($\gamma \neq \omega$), there exists a local run $\rho_T = (\nu_{in}, \nu_{out}, \{(\rho_i, \sigma_i)\}_{0 \le i < \gamma})$ on finite database $D_T \models$* **FD** *such that for* *every* $0 \leq i < \gamma$,

(i) for every expression $e = x_R.w$ *where* $\nu_i^*(e)$ *is defined, there exists expression* $e' = x_R.w'$ *where* $|w'| \leq h(T)$ *such that* $\nu_i^*(e) = \nu_i^*(e')$,

(ii) for all expressions $e, e' \in \mathcal{E}_T^+$ *of* τ_i*, if* $\nu_i^*(e)$ *and* $\nu_i^*(e')$ *are defined, then* $e \sim_{\tau_i} e'$ *iff* $\nu_i^*(e) = \nu_i^*(e')$*, and*

(iii) for $\delta = h(T_c)$ *if* $\sigma_i \in \{\sigma_{T_c}^o, \sigma_{T_c}^c\}$ *for some* $T_c \in child(T)$ *and* $\delta = 1$ *otherwise, for every expression* $e \in \mathcal{E}_T^+ - \{x_R.w | x \in \bar{x}^T, |w| > \delta\}$, $\nu_i^*(e)$ *is defined.*

Part (i), needed for technical reasons, says that for all values v in D_T, if v is the value of expression $x_R.w$, then v is also the value of an expression $x_R.w'$ where the length of w' is within $h(T)$. Part (ii) says, intuitively, that the equality types in the symbolic local run and the constructed local run are the same. Part (iii) states that for every $0 \leq i < \gamma$, at valuation ν_i, every expression e within δ steps of foreign key navigation from any variable x is defined in D_T. Since $\delta \geq 1$, this together with (ii) implies that for every condition π, $\tau_i \models \pi$ iff $D_T \models \pi(\nu_i)$. So if $\tilde{\rho}_T$ is accepted by some computation of a Büchi automaton $B(T, \eta)$ then ρ_T is also accepted by the same computation of $B(T, \eta)$.

We provide the proof of Lemma 44 in the remainder of the section. We first show that from each finite local symbolic run $\tilde{\rho}_T$, we can construct a *global isomorphism type* of $\tilde{\rho}_T$, which is essentially an equality type over the entire set of expressions in the symbolic instances of $\tilde{\rho}_T$. Then we show that the local run ρ_T and database D_T whose domain values are the equivalence classes of the global isomorphism type, satisfy the properties in Lemma 44.

Global isomorphism types. We prove Lemma 44 by constructing ρ_T and D_T from $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$ $(\gamma \neq \omega)$. We first introduce some additional notation.

Let $\mathcal{I}^+$ be the set of symbolic instances I_i of $\tilde{\rho}_T$ $(i < \gamma - 1)$ such that $+S^T(\bar{s}^T) \in \delta_{i+1}$ and $\hat{\tau}_i$ is not input-bound. Similarly let $\mathcal{I}^-$ be the set of symbolic instances I_j $(j < \gamma)$ such that $-S^T(\bar{s}^T) \in \delta_j$ and $\hat{\tau}_j$ is not input-bound. We define a one-to-one function `Retrieve` from $\mathcal{I}^-$ to $\mathcal{I}^+$ such that for every $I_i = \texttt{Retrieve}(I_j)$, $i < j$ and $\hat{\tau}_i = \hat{\tau}_j$. We say that I_j retrieves from I_i. As $\bar{c}_i \geq 0$ for every i, at least one mapping `Retrieve` always exists. Intuitively, `Retrieve` connects symbolic instance I_j to I_i such that I_j retrieves a tuple from S^T which has the same isomorphism type as a tuple inserted at I_i. For each $I_i = \texttt{Retrieve}(I_j)$, in the local run ρ_T we construct, valuations ν_i and ν_j have same values on variables $\bar{s}^T$. Here we ignore input-bound isomorphism types since these can be seen as part of the input isomorphism type: in ρ_T, instances having the same input-bound TS-isomorphism type have the same values on $\bar{s}^T$.

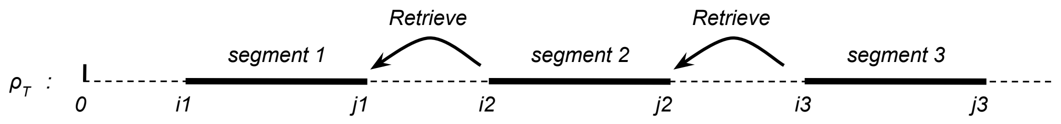


Figure 3.11. An illustration of a life cycle.

Recall that a *segment* $S = \{(I_i, \sigma_i)\}_{a \leq i \leq b}$ is a maximum consecutive subsequence of $\{(I_i, \sigma_i)\}_{0 \leq i < \gamma}$ such that σ_a is an internal service and for $a < i \leq b$, σ_i is opening service or closing service of child tasks of T. For our choice of the `Retrieve` relation, we define a *life cycle* $L = \{(I_i, \sigma_i)\}_{i \in J}$ as a maximum subsequence of $\{(I_i, \sigma_i)\}_{0 \leq i < \gamma}$ for $J \subseteq [0, \gamma)$ where for each pair of consecutive (I_a, σ_a) and (I_b, σ_b) in L where $a < b$, (I_a, σ_a) and (I_b, σ_b) are either in the same segment or $I_a = \texttt{Retrieve}(I_b)$ (illustrated in Figure 3.11). Note that a life cycle L is also a sequence of segments. From the definition of local symbolic runs, we can show the following properties for segments and life cycles:

Lemma 45. (i) *For every segment* $S = \{(I_i, \sigma_i)\}_{a \leq i \leq b}$*, for every* $i, j \in [a, b]$ *where* $i < j$*, for* $\bar{x} = \{x | x \in \bar{x}^T, x \not\sim_{\tau_i} \texttt{null}\}$*,* $\tau_i | \bar{x} = \tau_j | \bar{x}$*.* (ii) *For every life cycle* $L = \{(I_i, \sigma_i)\}_{i \in J}$*, for every* $i, j \in J$ *where* $i < j$*, for* $\bar{x} = \{x | x \in \bar{x}^T_{\text{in}} \cup \bar{s}^T, x \not\sim_{\tau_i} \texttt{null}\}$*,* $\tau_i | \bar{x} = \tau_j | \bar{x}$*.*

Next, for each symbolic instance I_i, we define the *pruned* isomorphism type $\lambda_i = (\mathcal{E}_i, \sim_i)$ of I_i as follows. Intuitively, each λ_i is obtained from τ_i by removing expressions with "long" navigation from variables. Formally, let $\mathcal{E}^+_T$ be the extended navigation set of τ_i and $\mathcal{E}^-_T = \mathcal{E}^+_T - \{x_R.w | x \in \bar{x}^T, |w| > \delta\}$, where $\delta = 1$ if T is a leaf task, otherwise $\delta = \max_{T_c \in \mathit{child}(T)} h(T_c)$. The choice of δ ensures that the remaining information in each λ_i

is sufficient for the consistency of keys and foreign keys within one single transition (i.e. an internal transition or a child task return). A *local expression* of I_i is a pair (i, e) where $e \in \mathcal{E}_T^-$, and we define $\mathcal{E}_i = \{(i, e) | e \in \mathcal{E}_T^-\}$ as the *local navigation set* of λ_i. We also define the *local equality type* $\sim_i$ of λ_i to be an equality type over $\mathcal{E}_i$ where $(i, e) \sim_i (i, e')$ iff $e \sim_{\tau_i} e'$, for every $e, e' \in \mathcal{E}_T^-$. Intuitively, the set $\mathcal{E}_i$ contains all expressions that are assigned with fresh values when the actual local run is constructed.

Then we define the global isomorphism type as follows. A global isomorphism type is a pair $\Lambda = (\mathcal{E}, \sim)$, where $\mathcal{E} = \bigcup_{0 \le i < \gamma} \mathcal{E}_i$ is called the *global navigation set* and $\sim$ is an equality type over $\mathcal{E}$ called *global equality type*. For each expression $e \in \mathcal{E}$, let $[e]$ denote its equivalence class with respect to $\sim$. The global equality type $\sim$ is constructed as follows:

1. **Initialization:** $\sim \leftarrow \bigcup_{0 \le i < \gamma} \sim_i$
2. **Chase:** Until convergence, merge two equivalence classes E and E' of $\sim$ if E and E' satisfy one of the following conditions:
   - **Segment-Condition:** For some segment $S = \{(I_i, \sigma_i)\}_{a \le i \le b}$, variable $x \in \bar{x}^T$ and $i, i' \in [a, b]$ where $x \not\sim_{\tau_i}$ `null` and $x \not\sim_{\tau_{i'}}$ `null`, $E = [(i, x)]$ and $E' = [(i', x)]$.
   - **Life-Cycle-Condition:** For some life cycle $L = \{(I_i, \sigma_i)\}_{i \in J}$, variable $x \in \bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$ and $i, i' \in J$ where $x \not\sim_{\tau_i}$ `null` and $x \not\sim_{\tau_{i'}}$ `null`, $E = [(i, x)]$ and $E' = [(i', x)]$.
   - **Input-Condition:** For some variable $x \in \bar{x}^T_{\mathsf{in}}$ and $i, i' \in [0, \gamma)$, $E = [(i, x)]$ and $E' = [(i', x)]$.
   - **FD-Condition:** For some local expressions $(i, e), (i', e')$ and attribute a where $(i, e) \sim (i', e')$, $E = [(i, e.a)]$ and $E' = [(i', e'.a)]$.

From the global isomorphism type Λ defined above, we construct ρ_T and D_T as follows. The domain of D_T is the set of equivalence classes of $\sim$. Each relation $R(id, a_1, \ldots, a_k)$ in D_T consists of all tuples $([(i, e)], [(i, e.a_1)], \ldots [(i, e.a_k)])$ for which $(i, e), (i, e.a_1), \ldots, (i, e.a_k) \in \mathcal{E}$. Note that the chase step guarantees that for all local expressions $(i, e), (i', e')$, if $(i, e.a), (i', e'.a) \in \mathcal{E}$ and $(i, e) \sim (i', e')$, then $(i, e.a) \sim (i', e'.a)$. It follows that $D_T \models$ **FD**. We next define

$\rho_T = (\nu_{in}, \nu_{out}, \{(\rho_i, \sigma_i)\}_{0 \le i < \gamma})$, where $\rho_i = (\nu_i, S_i)$. First, let $\nu_i(x) = [(i, x)]$ for $0 \le i < \gamma$, $\nu_{in} = \nu_0 | \bar{x}^T_{in}$, and $\nu_{out} = \perp$ if $\tau_{out} = \perp$ and $\nu_{out} = \nu_{\gamma-1} | \bar{x}^T_{\mathsf{out}}$ otherwise. Suppose that, as will be shown below, properties (i)-(iii) of Lemma 44 hold for D_T and the sequence $\{\nu_i\}_{0 \le i < \gamma}$ so defined. Note that (ii) and (iii) imply that the pre-and-post conditions of all services σ_i hold. Also, by construction, for every variable $x \in \bar{x}^T$ where $\nu_{i-1}(x) = \nu_i(x)$ is required by the transition under σ_i we always have $(i, x) \sim (i+1, x)$. Consider the sets $\{S_i\}_{0 \le i < \gamma}$. Recall the constraints imposed on sets by the definition of local run: $S_0 = \emptyset$, and for $0 < i < \gamma$ where δ_i is the set update of σ_i,

1. $S_i = S_{i-1} \cup \nu_{i-1}(\bar{s}^T)$ if $\delta_i = \{+S^T(\bar{s}^T)\}$,
2. $S_i = S_{i-1} - \nu_i(\bar{s}^T)$ if $\delta_i = \{-S^T(\bar{s}^T)\}$,
3. $S_i = (S_{i-1} \cup \{\nu_{i-1}(\bar{s}^T)\}) - \{\nu_i(\bar{s}^T)\}$ if $\delta_i = \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$, and
4. $S_i = S_{i-1}$ if $\delta_i = \emptyset$.

Note that the only cases that can make ρ_T invalid are those for which δ_i contains $-S^T(\bar{s}^T)$. Indeed, while a tuple can always be inserted, a tuple can be retrieved only if it belongs to S^T (or is simultaneously inserted as in case (3)). Thus, in order to show that the specified retrievals are possible, it is sufficient to prove the following.

Lemma 46. *Let $0 < i < \gamma$ be such that (1)-(4) hold for $\{S_j\}_{0 \le j < i}$. If $\delta_i = \{-S^T(\bar{s}^T)\}$ then $\nu_i(\bar{s}^T) \in S_{i-1}$. If $\delta_i = \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$ then either $\nu_i(\bar{s}^T) \in S_{i-1}$ or $\nu_i(\bar{s}^T) = \nu_{i-1}(\bar{s}^T)$.*

Proof. The key observations, which are easily checked by the construction of Λ, are the following:

(†) for every $k, k' \in [0, \gamma)$, if $\hat{\tau}_k$, $\hat{\tau}_{k'}$ are not input-bound and I_k and $I_{k'}$ are not in the same life cycle, then $\nu_k(\bar{s}^T) \neq \nu_{k'}(\bar{s}^T)$.

(‡) for every $k, k' \in [0, \gamma)$, if $\hat{\tau}_k$, $\hat{\tau}_{k'}$ are input-bound, $\nu_k(\bar{s}^T) = \nu_{k'}(\bar{s}^T)$ iff $\hat{\tau}_k = \hat{\tau}_{k'}$.

Now suppose that $0 < i < \gamma$, (1)-(4) hold for $\{S_j\}_{0 \le j < i}$, and $\delta_i = \{-S^T(\bar{s}^T)\}$. Suppose first that $\hat{\tau}_i$ is not input-bound. Let L be the life cycle to which I_i belongs, and $n < i$ be such that $I_n = \texttt{Retrieve}(I_i)$. By (†), $\nu_k(\bar{s}^T) \neq \nu_i(\bar{s}^T)$ for every $n < k < i$. Since (1)-(4) hold for all $j < i$, $\nu_n(\bar{s}^T) \in S_{i-1}$. By construction of Λ (specifically the Life-Cycle chase condition), $\nu_n(\bar{s}^T) = \nu_i(\bar{s}^T)$. Thus, $\nu_i(\bar{s}^T) \in S_{i-1}$. The case when $\delta_i = \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$ is similar.

Now suppose $\hat{\tau}_i$ is input-bound and $\delta_i = \{-S^T(\bar{s}^T)\}$. By definition of symbolic local run, $\bar{c}_{i-1}(\hat{\tau}_i) = 1$. Thus, there must exist a maximum $n < i$ such that $\hat{\tau}_n = \hat{\tau}_i$ and for which the transition under σ_n sets $\bar{c}_n(\hat{\tau}_i) = 1$. Since $\bar{c}_{i-1}(\hat{\tau}_i) = 1$ and n is maximal, there is no j, $n < j < i$ for which δ_j contains $-S^T(\bar{s}^T)$ and $\hat{\tau}_j = \hat{\tau}_i$. From the above and (‡) it easily follows that $\nu_n(\bar{s}^T) = \nu_i(\bar{s}^T)$ and $\nu_i(\bar{s}^T) \in S_{i-1}$. The case when $\delta_i = \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$ is similar. □

It remains to prove properties (i)-(iii) of Lemma 44. First, as $\delta \geq 1$ and $\delta \geq h(T_c)$ for every $T_c \in \mathit{child}(T)$, property (iii) is immediately satisfied. We next prove (i) and (ii).

Proof of property (i). We first introduce some additional notation. For each i and $(i, e) \in \mathcal{E}_i$, we denote by $[(i, e)]_i$ the equivalence class of (i, e) wrt $\sim_i$. And for $x \in \bar{x}^T$ we denote by $\texttt{Reach}_i(x, w)$ the unique equivalence class of $\sim_i$ reachable from $[(i, x_R)]_i$ by some navigation w (if such class exists). More precisely:

Definition 47. *For each $0 \leq i < \gamma$, we define the* navigation graph $G(\sim_i)$ *of the local equality type $\sim_i$ to be the labeled directed graph whose nodes are the equivalence classes of $\sim_i$ and where for each attribute a, there is an edge labeled a from E to F if there exist $e \in E$ and $f \in F$ such that $(i, e.a) \in \mathcal{E}_i$ and $e.a \sim_{\tau_i} f$. Note that for each E there is at most one outgoing edge labeled a. For $x \in \bar{x}^T, x \not\sim_i$ `null` and sequence of attributes w, we denote by $\texttt{Reach}_i(x, w)$ the unique equivalence class F of $\sim_i$ reachable from $[(i, x)]_i$ by a path in $G(\sim_i)$ whose sequence of edge labels spells w, if such exists, and the empty set otherwise.*

Example 48. *The graph $G(\sim_i)$ of the equality type in Example 32 is shown in Figure 3.12. In this $G(\sim_i)$, the node $\texttt{Reach}_i(y, AB)$ is the equivalence class $\{(i, x_R.A.B), (i, y_R.A.B), (i, 0)\}$.*

By our choice of $h(T)$ and our construction of the λ_i's, we can show that

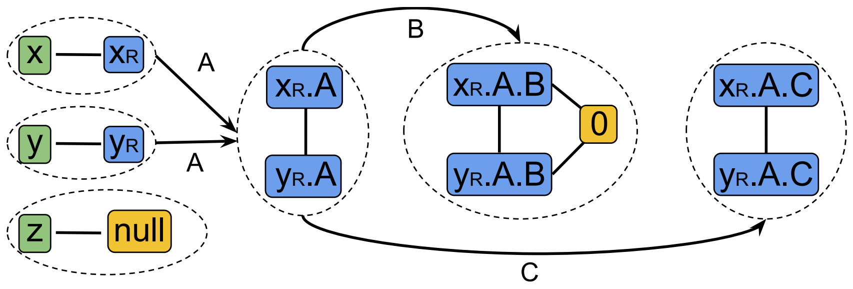


Figure 3.12. An illustration of the navigation graph $G(\sim_i)$.

Lemma 49. *For every $0 \le i < \gamma$ and expression $x_R.w$, if $\texttt{Reach}_i(x, w)$ is non-empty, then there exists an expression $x_R.\tilde{w}$ where $|\tilde{w}| < h(T)$ such that $\texttt{Reach}_i(x, w) = \texttt{Reach}_i(x, \tilde{w})$.*

Proof. It is sufficient to show that for each i, $|G(\sim_i)| < h(T)$, where $|G(\sim_i)|$ is the number of nodes in $G(\sim_i)$. Indeed, since there is a path from $[(i, x_R)]_i$ to $\texttt{Reach}_i(x, w)$ in $G(\sim_i)$, there must exist a simple such path, of length at most $|G(\sim_i)| < h(T)$.

To show that $|G(\sim_i)| < h(T)$, recall that $|G(\sim_i)|$ is bounded by the number of isomorphism types of $\sim_i$. Recall that $h(T) = 1 + |\bar{x}^T| \cdot F(\delta)$ where $F(n)$ is the maximum number of distinct paths of length at most n starting from any relation in the foreign key graph FK. By definition, for each variable x, the number of expressions $\{e | e = x_R.w, (i, e) \in \mathcal{E}_i\}$ is bounded by $F(\delta)$. Thus the number of equivalence classes of $\sim_i$ is at most $|\bar{x}^T| \cdot F(\delta) < h(T)$. So $|G(\sim_i)| < h(T)$. $\square$

Property (i) now follows from Lemma 49. Let $e = x_R.w$ be an expression for which $\nu_i^*(e)$ is defined. By construction, $\texttt{Reach}_i(x, w) \subseteq \nu_i^*(e)$. By Lemma 49, there exists $e' = x_R.w'$ where $|w'| < h(T)$ and $\texttt{Reach}_i(x, w') = \texttt{Reach}(x, w)$. It follows that $\nu_i^*(e')$ is defined and $\nu_i^*(e) \cap \nu_i^*(e') \neq \emptyset$. As $\nu_i^*(e)$ and $\nu_i^*(e')$ are equivalence classes of $\sim$, we have $\nu_i^*(e) = \nu_i^*(e')$, proving (i).

Proof of property (ii). To show property (ii), it is sufficient to show an invariant which implies property (ii) and is satisfied throughout the construction of Λ. For simplicity, we assume that the chase step in the construction of $\sim$ is divided into the following 3 phases.

- The *Segment Phase*. In this phase, we merge equivalence classes E and E' that satisfies either the Segment-Condition or the FD-condition.

- The *Life Cycle Phase*. In this phase, we merge equivalence classes E and E' that satisfies either the Life-Cycle-Condition or the FD-condition.
- The *Input Phase*. In this phase, we merge equivalence classes E and E' that satisfies either the Input-condition or the FD-condition.

It is easily seen that no chase step applies after the input phase. Thus, the above steps compute the complete chase.

For each equivalence class E of $\sim$, we let $i(E)$ be the set of indices $\{i|(i,e) \in E\}$ and for each $i \in i(E)$, we denote by $E|_i$ the projection of E on the navigation set $\mathcal{E}_i$. One can show that during the segment phase, for every E of $\sim$, $i(E)$ are indices within the same segment. During the life cycle phase, for every E of $\sim$, $i(E)$ are indices within the same life cycle. And during the input phase, $i(E)$ can be arbitrary indices.

The invariant is defined as follows.

Lemma 50. *(Invariant of Λ) Throughout the construction of Λ, for every equivalence class E of $\sim$, there exists variable $x \in \bar{x}^T$ and navigation w where $|w| \leq h(T)$, such that for every $i \in i(E)$, $E|_i = \texttt{Reach}_i(x, w)$.*

Lemma 50 implies that for each equivalence class E of $\sim$ and for each λ_i, E is a superset of at most one equivalence class of λ_i. So $(i,e) \sim (i,e')$ implies $(i,e) \sim_i (i,e')$ thus $\Lambda|\mathcal{E}_i = \lambda_i$ for every $0 \leq i < \gamma$, which implies property (ii) of Lemma 44.

Proof. We consider each step of the construction of the global equality type $\sim$. For the initialization step, the invariant holds by Lemma 49.

For the Chase steps, assume that the invariant is satisfied before merging two equivalence classes E and E'. For each equivalence class E of $\sim$, we denote by $x(E)$ and $w(E)$ the variable and the navigation for E as stated in Lemma 50. To show the invariant is satisfied after merging E and E', it is sufficient to show that there exists variable y and navigation u where $|u| \leq h(T)$ such that for every $i \in i(E)$, $E|_i = \texttt{Reach}_i(y, u)$ and for every $i \in i(E')$, $E'|_i = \texttt{Reach}_i(y, u)$.

Consider the segment phase. Suppose first that E and E' are merged due to the Segment-Condition. For simplicity, we let $x = x(E), x' = x(E'), w = w(E)$ and $w' = w(E')$. If

$E = [(i, y)]$ and $E' = [(i', y)]$ where i, i' are indices within the same segment S, then by the assumption, we have $(i, y) \in \texttt{Reach}_i(x, w)$, so $y \sim_{\tau_i} x_R.w$. As $i(E)$ are indices of a segment S, and by Lemma 45, we have that for every $j \in i(E)$, $y \sim_{\tau_j} x_R.w$, so $E|_j = \texttt{Reach}_j(x, w) = \texttt{Reach}_j(y, \epsilon)$. Similarly, we can show that for every $j \in i(E')$, $E'|_j = \texttt{Reach}_j(y, \epsilon)$.

Next suppose E and E' are merged due to the FD-condition. Thus, $E = [(i, e.a)]$ and $E' = [(i', e'.a)]$ where $(i, e) \sim (i', e')$. Let E^* be the equivalence class of $\sim$ that contains (i, e) and (i', e'). By the assumption, for $y = x(E^*)$ and $u = w(E^*)$, we have that $E^*|_i = \texttt{Reach}_i(y, u)$ so $(i, e) \in \texttt{Reach}_i(y, u)$. By Lemma 49, there exists navigation $\tilde{u}$ where $|\tilde{u}| < h(T)$ such that $\texttt{Reach}_i(y, u) = \texttt{Reach}_i(y, \tilde{u})$. So $(i, e.a) \in \texttt{Reach}_i(y, \tilde{u}.a)$. Then in E, by the hypothesis, we have $(i, e.a) \in \texttt{Reach}_i(x, w)$ so $\texttt{Reach}_i(y, \tilde{u}.a) = \texttt{Reach}_i(x, w)$. As $i(E)$ are indices of a segment S, and by Lemma 45, we have that for every $j \in i(E)$, for some relation R_1 and R_2, $y_{R_1}.\tilde{u}.a \sim_{\tau_j} x_{R_2}.w$ so $E|_j = \texttt{Reach}_j(x, w) = \texttt{Reach}_j(y, \tilde{u}.a)$. Similarly, we can show that for every $j \in i(E')$, $E'|_j = \texttt{Reach}_j(y, \tilde{u}.a)$. Therefore, the invariant is preserved during the segment phase.

Consider the life cycle phase. We can show that the invariant is again preserved, together with the following additional property: for each equivalence class E of $\sim$ produced in this phase, $x(E) \in \bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$. Suppose E and E' are merged due to the Life-Cycle Condition, where $E = [(i, y)]$, $E' = [(i', y)]$ and $y \in \bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$. We have that $E|_j = \texttt{Reach}_j(x, w) = \texttt{Reach}_j(y, \epsilon)$ for every $j \in i(E)$. Indeed, by Lemma 45 and because $i(E)$ are indices of some life cycle L, $x_R.w \sim_{\tau_i} y$ implies that $x_R.w \sim_{\tau_j} y$ for every index j of L. Similarly, $E'|_j = \texttt{Reach}_j(y, \epsilon)$ for every $j \in i(E')$. The case when E and E' are merged in this stage due to the FD-condition is similar to the above. Following similar analysis, we can show that the input phase also preserves the invariant together with the property that for every E produced at the input phase, $x(E) \in \bar{x}^T_{\mathsf{in}}$. This uses the fact that $\tau_i|\bar{x}^T_{\mathsf{in}} = \tau_{in}$ for every $0 \leq i < \gamma$. □

This completes the proof of Lemma 44.

3.5.5 Handling infinite local symbolic runs

Next we show that Lemma 44 can be extended to infinite periodic local symbolic runs, which together with finite runs are sufficient to represent accepted symbolic trees of runs by our VASS construction (see Lemma 65). Specifically, we show that we can extend the construction of the global isomorphism type to infinite periodic $\tilde{\rho}_T$, while producing only *finitely* many equivalence classes. This is sufficient to show that the corresponding database D_T is finite. We define periodic local symbolic runs next.

Definition 51. *A local symbolic run $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$ is periodic if $\gamma = \omega$ and there exists $n > 0$ and $0 < t \leq n$, such that for every $i \geq n$, the symbolic instances $I_i = (\tau_i, \bar{c}_i)$ and $I_{i-t} = (\tau_{i-t}, \bar{c}_{i-t})$ satisfy that $(\tau_i, \sigma_i) = (\tau_{i-t}, \sigma_{i-t})$ and $\bar{c}_i \geq \bar{c}_{i-t}$. The integer n and t are called the* offset *and* period *of $\tilde{\rho}_T$ respectively.*

The following is a consequence of Lemma 65, proven later in the section.

Corollary 52. *If there exists a symbolic tree of runs **Sym** accepted by $\mathcal{B}_\beta$, then there exists a symbolic tree of runs **Sym**$'$ accepted by $\mathcal{B}_\beta$ such that for every $\tilde{\rho}_T \in$ **Sym**, $\tilde{\rho}_T$ is finite or periodic.*

The above corollary indicates that for verification, it is sufficient to consider only finite and periodic $\tilde{\rho}_T$. So what we need to prove is:

Lemma 53. *For every periodic local symbolic run $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \omega})$, there exists a local run $\rho_T = (\nu_{in}, \nu_{out}, \{(\rho_i, \sigma_i)\}_{0 \leq i < \omega})$ on finite database $D_T \models$ **FD** such that for every $i \geq 0$,*

(i) for every expression $e = x_R.w$ where $\nu_i^*(e)$ is defined, there exists expression $e' = x_R.w'$ where $|w'| \leq h(T)$ such that $\nu_i^*(e) = \nu_i^*(e')$,

(ii) for all expressions $e, e' \in \mathcal{E}_T^+$ of τ_i, if $\nu_i^*(e)$ and $\nu_i^*(e')$ are defined, then $e \sim_{\tau_i} e'$ iff $\nu_i^*(e) = \nu_i^*(e')$, and

(iii) for $\delta = h(T_c)$ if $\sigma_i \in \{\sigma_{T_c}^o, \sigma_{T_c}^c\}$ for some $T_c \in child(T)$ and $\delta = 1$ otherwise, for every expression $e \in \mathcal{E}_T^+ - \{x_R.w | x \in \bar{x}^T, |w| > \delta\}$, $\nu_i^*(e)$ is defined.

Intuitively, if we directly apply the construction of ρ_T and D_T from Lemma 44 in the case of finite $\tilde{\rho}_T$, then each life cycle with non-input-bound TS-isomorphism types would be assigned with distinct sets of values, which could lead to an infinite D_T. However, for any two disjoint life cycles L_1 and L_2, reusing the same values in L_1 and L_2 does not cause any conflict. And in particular, if L_1 and L_2 are identical on the sequence of τ_i's and σ_i's, they can share exactly the same set of values.

Thus at a high level, our goal is to show that any periodic local symbolic run $\tilde{\rho}_T$ can be partitioned into finitely many subsets of identical life cycles with disjoint timespans. Unfortunately, this is generally not true if we pick the `Retrieve` function arbitrarily (recall that `Retrieve` defines the set of life cycles). This is because an arbitrary `Retrieve` may yield life cycles whose timespans have unbounded length. If the timespans overlap, it is impossible to separate the life cycles into finitely many subsets of life cycles with disjoint timespans. So instead of picking an arbitrary `Retrieve` as in the finite case, we show that for periodic $\tilde{\rho}_T$ we can construct `Retrieve` such that the timespan of each life cycle has bounded length. This implies that we can partition the life cycles into finitely many subsets of identical life cycles with disjoint timespans, as desired. Finally we show that given the partition, we can construct the local run ρ_T together with a finite D_T.

We first define the equivalence relation between life cycles.

Definition 54. *Segments* $S_1 = \{(I_i, \sigma_i)\}_{a_1 \le i \le b_1}$ *and* $S_2 = \{(I_i, \sigma_i)\}_{a_2 \le i \le b_2}$ *are equivalent, denoted as* $S_1 \equiv S_2$*, if* $\{(\tau_i, \sigma_i)\}_{a_1 \le i \le b_1} = \{(\tau_i, \sigma_i)\}_{a_2 \le i \le b_2}$.

Recall that $\mathcal{I}^+$ (and $\mathcal{I}^-$) are the sets of symbolic instances inserting (and retrieving) non-input-bound TS-isomorphism types respectively. We define that

Definition 55. *A segment* $S = \{(I_i, \sigma_i)\}_{a \le i \le b}$ *is* static *if* $I_a \in \mathcal{I}^-$, $I_b \in \mathcal{I}^+$ *and* $\tau_a|\bar{s}^T = \tau_b|\bar{s}^T$. *A segment* S *is called* dynamic *if it is not static.*

When we compare two life cycles L_1 and L_2, we can ignore their static segments since they do not change the content of S^T. We define equivalence of two life cycles as follows.

Definition 56. *For life cycle L, let $dyn(L) = \{S_i\}_{1 \le i \le k}$ be the sequence of dynamic segments of L. Two life cycles L_1 and L_2 are equivalent, denoted as $L_1 \equiv L_2$, if $|dyn(L_1)| = |dyn(L_2)|$ and for $dyn(L_1) = \{S_i^1\}_{1 \le i \le k}$ and $dyn(L_2) = \{S_i^2\}_{1 \le i \le k}$, for every $1 \le i \le k$, $S_i^1 \equiv S_i^2$.*

Note that for each life cycle L, the number of dynamic segments within L is bounded by $|\bar{s}^T|$ since within L, each variable in $\bar{s}^T$ is written at most once by returns of child tasks of T. For a task T, as the number of T-isomorphism types is bounded, the number of services is bounded and the length of a segment is bounded because each subtask can be called at most once, the number of equivalence classes of segments is bounded. And since the number of dynamic segments is bounded within the same life cycle, the number of equivalence classes of life cycles is also bounded. Thus,

Lemma 57. *The equivalence relation $\equiv$ on life cycles has finite index.*

Our next step is to show that one can define a `Retrieve` function so that all life cycles have bounded timespans. The timespan of a life cycle is defined as follows:

Definition 58. *The timespan of a life cycle L, denoted by $sp(L)$, is an interval $[a, b]$ where a is the index of the first symbolic instance of the first dynamic segment of L and b is the index of the last symbolic instance of the last dynamic segment.*

Consider an equivalence class $\mathcal{L}$ of life cycles. Suppose that for each $L \in \mathcal{L}$, the length of $sp(L)$ is bounded by some constant m. Then we can further partition $\mathcal{L}$ into m subsets $\mathcal{L}_0, \ldots, \mathcal{L}_{m-1}$ of life cycles with disjoint timespan by assigning each $L \in \mathcal{L}$ where $sp(L) = [a, b]$ to the subset $\mathcal{L}_k$ where $k = a \bmod m$.

We next show how to construct the function `Retrieve`. In particular, we construct a periodic `Retrieve` such that there is a short gap between each pair of inserting and retrieving instances. This is done in several steps, illustrated in Figure 3.13. Recall that n and t are the offset and the period of $\tilde{\rho}_T$.

1. Initialize `Retrieve` to be an arbitrary one-to-one mapping with domain $\{I_j | I_j \in \mathcal{I}^-, 0 \le$

$j \leq n\}$ such that for every $I_i = \texttt{Retrieve}(I_j)$, $i < j$ and $\hat{\tau}_i = \hat{\tau}_j$ (recall that $\hat{\tau}_i = \tau_i|\bar{x}_{\text{in}}^T \cup \bar{s}^T$).

2. For every $j \in [n+1, n+t]$, for $j' = j - t$ and for i' being the index where $I_{i'} = \texttt{Retrieve}(I_{j'})$,

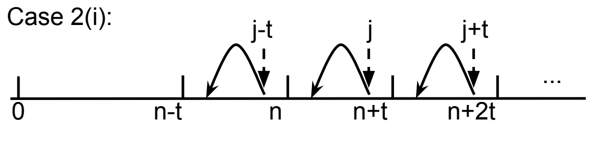
   (i) if $i' \in [n-t+1, n]$, then for $i = i'+t$, let $\texttt{Retrieve} \leftarrow \texttt{Retrieve}[I_{j+k\cdot t} \mapsto I_{i+k\cdot t}|k \geq 0]$, otherwise

   (ii) if $i' \in [0, n-t]$, then we pick $i \in [n-t+1, n]$ satisfying that $I_i \in \mathcal{I}^+$, $\hat{\tau}_i = \hat{\tau}_j$ and I_i is currently not in the range of $\texttt{Retrieve}$. Then we let $\texttt{Retrieve} \leftarrow \texttt{Retrieve}[I_{j+k\cdot t} \mapsto I_{i+k\cdot t}|k \geq 0]$.

At step 2 for the case $i' \in [0, n-t]$, the i that we picked always exists for the following reason. For every TS-isomorphism type $\hat{\tau}$, let

- $M_{\hat{\tau}}^-$ be the number of symbolic instances in $\mathcal{I}^-$ with TS-isomorphism type $\hat{\tau}$ and indices in $[n-t+1, n]$ that retrieves from symbolic instances with indices in $[0, n-t]$, and
- $M_{\hat{\tau}}^+$ be the number of symbolic instances in $\mathcal{I}^+$ with TS-isomorphism type $\hat{\tau}$ and indices in $[n-t+1, n]$ that is NOT retrieved by symbolic instances with indices in $[n-t+1, n]$.

We have $M_{\hat{\tau}}^+ - M_{\hat{\tau}}^- = \bar{c}_n(\hat{\tau}) - \bar{c}_{n-t}(\hat{\tau}) \geq 0$. So for every $I_{i'} = \texttt{Retrieve}(I_{j'})$ where $j' \in [n-t+1, n]$ and $i' \in [0, n-t]$, we can always find a unique $i \in [n-t+1, n]$ such that $I_i \in \mathcal{E}^+$, $\hat{\tau}_i = \hat{\tau}_{j'} = \hat{\tau}_{i'}$ and I_i is not retrieved by any retrieving instances with indices in $[n-t+1, n]$.



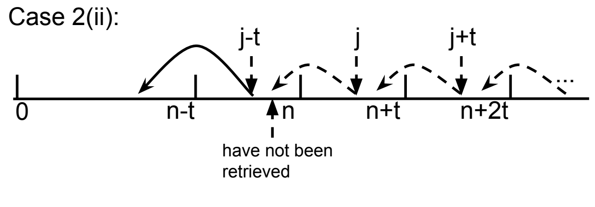


Figure 3.13. Construction of $\texttt{Retrieve}$

Let us fix the function $\texttt{Retrieve}$ constructed above. We first show the following:

Lemma 59. *For every periodic $\tilde{\rho}_T$, and $j > n$, $I_i = \texttt{Retrieve}(I_j)$ implies that $j - i \leq 2t$ and $I_{i+t} = \texttt{Retrieve}(I_{j+t})$.*

Proof. By construction, for every $I_i = \texttt{Retrieve}(I_j)$ where $j > i > n$, $I_{i+t} = \texttt{Retrieve}(I_{j+t})$. And it is also guaranteed that for the indices i and j, either (1) i and j are both in the same range $[n + tk + 1, n + t(k + 1)]$ for some $k \geq 0$, or (2) $i \in [n + tk + 1, n + t(k + 1)]$ and $j \in [n + t(k + 1) + 1, n + t(k + 2)]$ for some $k \geq 0$. In both cases, $j - i \leq 2t$. $\square$

For every life cycle L, for every pair of consecutive dynamic segments S and S', we denote by $\mathit{gap}(S, S')$ the number of static segments in between S and S'. To show that $sp(L)$ is bounded, it is sufficient to show that $\mathit{gap}(S, S')$ is bounded for every pair of consecutive dynamic segments S and S'. For every segment S, we denote by $a(S)$ the index of the first symbolic instance of S. For every segment S where $a(S) > n$, we let $p(S) = (a(S) - n - 1) \bmod t$.

For every pair of consecutive dynamic segments S and S' and by periodicity of Retrieve, there are no two static segments T and T' in L in between S and S' such that $a(S) < a(T) < a(T') < a(S')$ and $p(T) = p(T')$. Thus in L, the number of static segments in between S and S' is at most $n + t$. Then by Lemma 59, the number of symbolic instances in between any pair of consecutive segments is bounded by $\max(2t, n)$ so $\mathit{gap}(S, S') \leq (n + t) \cdot \max(2t, n + t)$. And as the number of dynamic segments in L is bounded by $|\bar{s}^T|$ and the length of each segment is at most $2|\mathit{child}(T)|$, it follows that:

Lemma 60. *For every periodic local symbolic run $\tilde{\rho}_T$ and life cycle L of $\tilde{\rho}_T$, $|sp(L)|$ is bounded by $m = (n + t) \cdot \max(2t, n + t) \cdot (|\bar{s}^T| + 1) \cdot 2|\mathit{child}(T)|$.*

So for a possibly infinite set of life cycles $\mathcal{L}$ where $|sp(L)| \leq m$ for each $L \in \mathcal{L}$, $\mathcal{L}$ can be partitioned into sets $\mathcal{L}_0, \ldots, \mathcal{L}_{m-1}$ by assigning each life cycle $L \in \mathcal{L}$ where $sp(L) = [a, b]$ to the set $\mathcal{L}_{a \bmod m}$. So for every $\mathcal{L}_i$ and two distinct L_1, L_2 in $\mathcal{L}_i$ where $sp(L_1) = [a_1, b_1]$ and $sp(L_2) = [a_2, b_2]$, we have $a_1 \neq a_2$. Assume $a_1 < a_2$. Then as $a_1 \equiv a_2 \pmod m$, $a_2 - a_1 \geq m$. And since $b_1 - a_1 + 1 < m$, L_1 and L_2 are disjoint. Thus, given Lemma 57 and Lemma 60, we have

Lemma 61. *Every local symbolic run $\tilde{\rho}_T$ can be partitioned into finitely many subsets of life cycles such that for each subset $\mathcal{L}$, if $L_1 \in \mathcal{L}$, $L_2 \in \mathcal{L}$ and $L_1 \neq L_2$ then $L_1 \equiv L_2$ and $sp(L_1) \cap sp(L_2) = \emptyset$.*

Next, we show how we can construct the local run ρ_T and finite database D_T from $\tilde{\rho}_T$ using the partition. We first construct a global isomorphism type $\Lambda = (\mathcal{E}, \sim)$ of $\tilde{\rho}_T$ using the approach for the finite case. Then we merge equivalent segments in Λ as follows to obtain a new global isomorphism type with finitely many equivalence classes. To merge two equivalent segments $S_1 = \{(I_i, \sigma_i)\}_{a_1 \leq i \leq a_1+l}$ and $S_2 = \{(I_i, \sigma_i)\}_{a_2 \leq i \leq a_2+l}$, first for every $0 \leq i \leq l$ and for every $x \in \bar{x}^T$, we merge the equivalence classes $[(a_1 + i, x)]$ and $[(a_2 + i, x)]$ of $\sim$. Then we apply the chase step (i.e. the FD-condition) to make sure the resulting database satisfies **FD**.

The new Λ is constructed as follows. For every two segments $S_1 = \{(I_i, \sigma_i)\}_{a \leq i \leq b}$ and $S_2 = \{(I_i, \sigma_i)\}_{c \leq i \leq d}$, we define that S_1 precedes S_2, denote by $S_1 \prec S_2$, if $b < c$. For each subset $\mathcal{L}$ and for each pair of life cycles $L_1, L_2 \in \mathcal{L}$ where $dyn(L_1) = \{S_i^1\}_{1 \leq i \leq k}$ and $dyn(L_2) = \{S_i^2\}_{1 \leq i \leq k}$,

- for $1 \leq i \leq k$, merge S_i^1 and S_i^2,
- for $1 \leq i < k$, for every static segments $S_1 \subseteq L_1$ and $S_2 \subseteq L_2$ where $S_i^1 \prec S_1 \prec S_{i+1}^1$, $S_i^2 \prec S_2 \prec S_{i+1}^2$ and $S_1 \equiv S_2$, merge S_1 and S_2, and
- for every pair of static segments $S_1 \subseteq L_1$ and $S_2 \subseteq L_2$ where $S_k^1 \prec S_1$, $S_k^2 \prec S_2$ and $S_1 \equiv S_2$, merge S_1 and S_2.

Finally, ρ_T and D_T are constructed following the same approach as in the finite case. In the above construction, as the number of subsets of life cycles is finite, and for each $\mathcal{L}$, the number of dynamic segments is bounded and the number of equivalence classes of static segments is bounded, the number of equivalence classes of Λ is also finite so D_T is finite.

By an analysis similar to the finite case, we can show that ρ_T and D_T satisfy property (i)-(iii) in Lemma 53 and $D_T \models$ **FD**. In particular, to show property (ii), we can show the same invariant as in Lemma 50, the invariant holds because every pair of merged segments are equivalent.

Finally, to show Lemma 53, it remains to show that ρ_T is a valid local run. Similar to the finite case, it is sufficient to show that

Lemma 62. *For every $i \geq 0$, if $\delta_i = \{-S^T(\bar{s}^T)\}$ then $\nu_i(\bar{s}^T) \in S_{i-1}$. If $\delta_i = \{+S^T(\bar{s}^T), -S^T(\bar{s}^T)\}$ then either $\nu_i(\bar{s}^T) \in S_{i-1}$ or $\nu_i(\bar{s}^T) = \nu_{i-1}(\bar{s}^T)$.*

Proof. The following can be easily checked by the construction of Λ:

(i) for every pair of distinct life cycles L and L' where $sp(L) \cap sp(L') \neq \emptyset$, for every $I_k \in L$ and $I_{k'} \in L'$, if $\hat{\tau}_k$, $\hat{\tau}_{k'}$ are not input-bound then $\nu_k(\bar{s}^T) \neq \nu_{k'}(\bar{s}^T)$, and

(ii) for every pair of life cycles L and L' where $sp(L) \cap sp(L') = \emptyset$, if $I_i, I_j \in L$, $I_j = \texttt{Retrieve}(I_i)$, $\hat{\tau}_i$ is not input-bound, $I_k \in L'$ for $j < k < i$ and $\nu_k(\bar{s}^T) = \nu_i(\bar{s}^T) = \nu_j(\bar{s}^T)$, then I_k is contained in a static segment of L'.

(iii) for every $k, k' \geq 0$, if $\hat{\tau}_k$, $\hat{\tau}_{k'}$ are input-bound, $\nu_k(\bar{s}^T) = \nu_{k'}(\bar{s}^T)$ iff $\hat{\tau}_k = \hat{\tau}_{k'}$.

Consider the case when $\delta_i = \{-S^T(\bar{s}^T)\}$ and $\hat{\tau}_i$ is not input-bound. Let $I_j = \texttt{Retrieve}(I_i)$ and L be the life cycle that contains I_i. Consider I_k where $j < k < i$ and let L' be the life cycle containing I_k. If $sp(L) \cap sp(L') \neq \emptyset$, by (i), $\nu_i(\bar{s}^T) \neq \nu_k(\bar{s}^T)$. If $sp(L) \cap sp(L') = \emptyset$, by (ii), the segment containing I_k is static, so it does not change S^T. Thus, for every segment S between I_j and I_i, the tuple $\nu_i(\bar{s}^T)$ remains in S^T after S. So $\nu_i(\bar{s}^T) \in S_{i-1}$. The case when $\delta_i = \{-S^T(\bar{s}^T), +S^T(\bar{s}^T)\}$ is similar.

The proof for the case when $\hat{\tau}_i$ is input-bound is the same as the proof for Lemma 46. □

This completes the proof of Lemma 53.

3.5.6 Handling symbolic trees of runs

Finally, we show Lemma 42 by providing a recursive construction of a tree of runs ***Tree*** and database D from any symbolic tree of runs ***Sym*** where all local symbolic runs are either finite or periodic, using Lemmas 44 and 53. Intuitively, the construction simply applies the two lemmas to each node $\tilde{\rho}_T$ of ***Sym*** to obtain a local run ρ_T with a local database D_T. Then the local runs and databases are combined into a tree of local runs recursively by renaming the values in each ρ_T and D_T in a bottom-up manner, reflecting the communication among local runs via input and return variables.

Formally, we first define recursively the construction function F where $F(\boldsymbol{Sym}_T) = (\boldsymbol{Tree}_T, D_T)$ where $\boldsymbol{Sym}_T$ is a subtree of $\boldsymbol{Sym}$ and $(\boldsymbol{Tree}_T, D_T)$ are the resulting subtree of local runs and database instance. F is defined as follows.

If T is a leaf task, then $\boldsymbol{Sym}_T$ contains a single local symbolic run $\tilde{\rho}_T$. We define that $F(\boldsymbol{Sym}_T) = F(\tilde{\rho}_T) = (\rho_T, D_T)$ where ρ_T and D_T are the local run and database instance shown to exist in Lemmas 44 and 53 corresponding to $\tilde{\rho}_T$.

If T is a non-leaf task where the root of $\boldsymbol{Sym}_T$ is $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$, then we first let $(\rho_T, D_{\texttt{root}}) = F(\tilde{\rho}_T)$. Next, let $J = \{i | \sigma_i = \sigma^o_{T_c}, T_c \in \mathit{child}(T)\}$. For every $i \in J$, we denote by $\boldsymbol{Sym}_i$ the subtree rooted at the child of $\tilde{\rho}_T$ where the edge connecting it with $\tilde{\rho}_T$ is labeled i and let $\tilde{\rho}_i$ be the root of $\boldsymbol{Sym}_i$. We denote by $(\boldsymbol{Tree}_i, D_i) = F(\boldsymbol{Sym}_i)$ and by ρ_i the local run at the root of $\boldsymbol{Tree}_i$. From the construction in Lemmas 44 and 53, we assume that the domains of $D_{\texttt{root}}$ and the D_i's are equivalence classes of local expressions. We first define the renaming function r whose domain is $\bigcup_{i \in J} \texttt{adom}(D_i)$ as follows.

1. Initialize r to be the identity function.
2. For every $i \in J$, for every expression $x_R.w$ where $x \in \bar{x}^{T_c}_{\mathsf{in}}$ and $\nu^*_{in}(x_R.w)$ is defined, for $y = f_{in}(x)$, let $r \leftarrow r[\nu^*_{in}(x_R.w) \mapsto \nu^*_i(y_R.w)]$. Note that ν^*_{in} is defined wrt ν_{in} of ρ_i and D_i and ν^*_i is defined wrt ν_i of ρ_T and $D_{\texttt{root}}$. And we shall see next that for every such $x_R.w$, if $\nu^*_{in}(x_R.w)$ is defined, then $\nu^*_i(y_R.w)$ is also defined.
3. For every $i \in J$ where $\tilde{\rho}_i$ is a returning local symbolic run where the index of the corresponding $\sigma^c_{T_c}$ in $\tilde{\rho}_T$ is j, for every expression $x_R.w$ where $x \in \bar{x}^{T_c}_{\mathsf{out}}$ and $\nu^*_{out}(x_R.w)$ is defined, for $y = f_{out}(x)$, let $r \leftarrow r[\nu^*_{out}(x_R.w) \mapsto \nu^*_j(y_R.w)]$.

We denote by $r(D)$ the database instance obtained by replacing each value $v \in \mathit{dom}(r)$ in D with $r(v)$ and denote by $r(\boldsymbol{Tree})$ the tree of runs obtained by replacing each value $v \in \mathit{dom}(r)$ in $\boldsymbol{Tree}$ with $r(v)$.

Then if $\tilde{\rho}_T$ is finite, we define $F(\textbf{\textit{Sym}}_T) = (\textbf{\textit{Tree}}_T, D_T)$ where $D_T = D_{\texttt{root}} \cup \bigcup_{i \in J} r(D_i)$ and $\textbf{\textit{Tree}}_T$ is obtained from $\textbf{\textit{Sym}}_T$ by replacing the root of $\textbf{\textit{Sym}}_T$ with ρ_T and each subtree $\textbf{\textit{Sym}}_i$ with $r(\textbf{\textit{Tree}}_i)$.

If $\tilde{\rho}_T$ is periodic where the period is t and the loop starts with index n, we define $F(\textbf{\textit{Sym}}_T) = (\textbf{\textit{Tree}}_T, D_T)$ where $D_T = D_{\texttt{root}} \cup \bigcup_{i \in J, i < n} r(D_i)$ and $\textbf{\textit{Tree}}_T$ is obtained from $\textbf{\textit{Sym}}_T$ by replacing the root of $\textbf{\textit{Sym}}_T$ with ρ_T and each subtree $\textbf{\textit{Sym}}_i$ with $r(\textbf{\textit{Tree}}_{i'})$, where $i' = i$ if $i < n$ otherwise $i' = n + (i - n) \mod t$.

To prove the correctness of the construction, we first need to show that for every $\textbf{\textit{Sym}}_T$ and $(\textbf{\textit{Tree}}_T, D_T) = F(\textbf{\textit{Sym}}_T)$, D_T is a finite database satisfying **FD** and $\textbf{\textit{Tree}}_T$ is a valid tree of runs over D_T. Let $\tilde{\rho}_T$ and ρ_T be the root of $\textbf{\textit{Sym}}_T$ and $\textbf{\textit{Tree}}_T$ respectively. We show the following:

Lemma 63. *For every symbolic tree of runs $\textbf{\textit{Sym}}_T$ where $(\textbf{\textit{Tree}}_T, D_T) = F(\textbf{\textit{Sym}}_T)$, D_T is a finite database satisfying* **FD**, *$\textbf{\textit{Tree}}_T$ is a valid tree of runs over D_T, and (ρ_T, D_T) satisfies properties (i)-(iii) in Lemma 44 and 53.*

Proof. We use a simple induction. For the base case, where T is a leaf task, the lemma holds trivially. For the induction step, assume that for each $i \in J$, D_i is finite and satisfies **FD**, $\textbf{\textit{Tree}}_i$ is a valid tree of runs over D_i, and (ρ_i, D_i) satisfies property (i)-(iii).

For each $i \in J$, where $\tilde{\rho}_i$ is a local symbolic run of task $T_c \in child(T)$, we first consider the connection between $\tilde{\rho}_i$ and $\tilde{\rho}_T$ via input variables. As ρ_i satisfies properties (i) and (ii), for every expressions $x_R.w$ and $x'_{R'}.w'$ in the input isomorphism type τ_{in} of $\tilde{\rho}_i$, if $\nu^*_{in}(x_R.w)$ and $\nu^*_{in}(x'_{R'}.w')$ are defined, then $\nu^*_{in}(x_R.w) = \nu^*_{in}(x'_{R'}.w')$ iff $x_R.w \sim_{\tau_{in}} x'_{R'}.w'$. And by definition of symbolic tree of runs, we have that $\tau_{in} = f_{in}^{-1}(\tau_i)|(\bar{x}^{T_c}_{\mathsf{in}}, h(T_c))$. So for $y = f_{in}(x)$ and $y' = f_{in}(x')$, $\nu^*_{in}(x_R.w) = \nu^*_{in}(x'_{R'}.w')$ iff $y_R.w \sim_{\tau_i} y'_{R'}.w'$. Then as ρ_T satisfies (ii) and (iii), $\nu^*_i(y_R.w)$ and $\nu^*_i(y'_{R'}.w')$ are defined and $\nu^*_i(y_R.w) = \nu^*_i(y'_{R'}.w')$ iff $y_R.w \sim_{\tau_i} y'_{R'}.w'$ so $\nu^*_i(y_R.w) = \nu^*_i(y'_{R'}.w')$ iff $\nu^*_{in}(x_R.w) = \nu^*_{in}(x'_{R'}.w')$.

If $\tilde{\rho}_i$ is returning, using the same argument as above, we can show the following. Let j

be the index of the corresponding returning service $\sigma^c_{T_c}$. Let f be the function where $f(x) = \begin{cases} f_{in}(x), x \in \bar{x}^{T_c}_{\mathsf{in}} \\ f_{out}(x), x \in \bar{x}^{T_c}_{\mathsf{out}} \end{cases}$ and let ν be the valuation where $\nu(x) = \begin{cases} \nu_{in}(x), x \in \bar{x}^{T_c}_{\mathsf{in}} \\ \nu_{out}(x), x \in \bar{x}^{T_c}_{\mathsf{out}} \end{cases}$, where ν_{in} and ν_{out} are the input and output valuation of ρ_i. For all expressions $x_R.w$ and $x'_{R'}.w'$ where $x, x' \in \bar{x}^{T_c}_{\mathsf{out}} \cup \bar{x}^{T_c}_{\mathsf{in}}$, if $\nu^*(x_R.w)$ and $\nu^*(x'_{R'}.w')$ are defined, then for $y = f(x)$ and $y' = f(x')$, $\nu^*_j(y_R.w)$ and $\nu^*_j(y'_{R'}.w')$ are also defined and $\nu^*_j(y_R.w) = \nu^*_j(y'_{R'}.w')$ iff $\nu^*(x_R.w) = \nu^*(x'_{R'}.w')$.

Given this, after renaming, $D_{\mathtt{root}}$ and $r(D_i)$ can be combined consistently. Also, one can easily check that ***Tree***$_T$ is a valid tree of runs where (ρ_T, D_T) satisfies properties (i)-(iii) and $D_T \models$ **FD**. And D_T is a finite database because it is the union of $D_{\mathtt{root}}$ and finitely many $r(D_i)$'s and by the hypothesis, $D_{\mathtt{root}}$ and the D_i's are finite. $\square$

Finally, to complete the proof of correctness of the construction, we note:

Lemma 64. *For every full symbolic tree of runs* ***Sym*** *where all local symbolic runs in* ***Sym*** *are either finite or periodic, for* $($***Tree***$, D) = F($***Sym***$)$ *and every HLTL-FO property* φ_f*,* ***Sym*** *is accepted by* $\mathcal{B}_\varphi$ *iff* ***Tree*** *is accepted by* $\mathcal{B}_\varphi$ *on* D*.*

The above follows immediately from the fact that by construction, for every task T and local symbolic run $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$ in ***Sym*** where the corresponding local run in ***Tree*** is $\rho_T = (\nu_{in}, \nu_{out}, \{(\rho_i, \sigma_i)\}_{0 \le i < \gamma})$, for every condition π over $\bar{x}^T$ and $0 \le i < \gamma$, $\tau_i \models \pi$ iff $D \models \pi(\nu_i)$.

This completes the proof of Lemma 42, and the only-if part of Theorem 39.

3.5.7 Symbolic Verification

In view of Theorem 39, we can now focus on the problem of checking the existence of a symbolic tree of runs satisfying a given HLTL-FO property. To begin, we define a notion that captures the functionality of each task and allows a modular approach to the verification algorithm. Let φ_f be an HLTL-FO formula over Γ, and recall the automaton $\mathcal{B}_\varphi$ and associated notation from Section 3.3. We consider the relation $\mathcal{R}_T$ between input and outputs of each task, defined by its symbolic runs that satisfy a given truth assignment β to the formulas in Φ_T. More specifically, we denote by $\mathcal{H}_T$ the restriction of $\mathcal{H}$ to T and its descendants, and Γ_T the corresponding HAS, with precondition *true*. The relation $\mathcal{R}_T$ consists of the set of triples $(\tau_{in}, \tau_{out}, \beta)$ for which there exists a symbolic tree of runs $\textbf{\textit{Sym}}_T$ of $\mathcal{H}_T$ such that:

- β is a truth assignment to Φ_T
- $\textbf{\textit{Sym}}_T$ is accepted by $\mathcal{B}_\beta$
- the root of $\textbf{\textit{Sym}}_T$ is $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$

Note that there exists a symbolic tree of runs ***Sym*** over Γ satisfying φ_f iff $(\tau_{in}, \perp, \beta) \in \mathcal{R}_{T_1}$ for some τ_{in} satisfying the precondition of Γ, and $\beta(\varphi_f) = 1$. Thus, if $\mathcal{R}_T$ is computable for every T, then satisfiability of φ_f by some symbolic tree of runs over Γ is decidable, and yields an algorithm for model-checking HLTL-FO properties of HAS's.

We next describe an algorithm that computes the relations $\mathcal{R}_T(\tau_{in}, \tau_{out}, \beta)$ recursively. The algorithm uses as a key tool Vector Addition Systems with States (VASS) [18, 69], which we review next.

A VASS $\mathcal{V}$ is a pair (Q, A) where Q is a finite set of *states* and A is a finite set of *actions* of the form $(p, \bar{a}, q)$ where $\bar{a} \in \mathbb{Z}^d$ for some fixed $d > 0$, and $p, q \in Q$. A run of $\mathcal{V} = (Q, A)$ is a finite sequence $(q_0, \bar{z}_0) \ldots (q_n, \bar{z}_n)$ where $\bar{z}_0 = \bar{0}$ and for each $i \geq 0$, $q_i \in Q$, $\bar{z}_i \in \mathbb{N}^d$, and $(q_i, \bar{a}, q_{i+1}) \in A$ for some $\bar{a}$ such that $\bar{z}_{i+1} = \bar{z}_i + \bar{a}$. We will use the following decision problems related to VASS.

- *State Reachability*: For given states $q_0, q_f \in Q$, is there a run $(q_0, \bar{z}_0) \ldots (q_n, \bar{z}_n)$ of $\mathcal{V}$ such that $q_n = q_f$?
- *State Repeated Reachability*: For given states $q_0, q_f \in Q$, is there a run $(q_0, \bar{z}_0) \ldots (q_m, \bar{z}_m) \ldots (q_n, \bar{z}_n)$ of $\mathcal{V}$ such that $q_m = q_n = q_f$ and $\bar{z}_m \leq \bar{z}_n$?

Both problems are known to be EXPSPACE-complete [91, 107, 69]. In particular, [69] shows that for a n-states, d-dimensional VASS where every dimension of each action has constant size, the state repeated reachability problem can be solved in $O((\log n)2^{c \cdot d \log d})$ non-deterministic space for some constant c. The state reachability problem has the same complexity.

VASS construction. Let T be a task, and suppose that relations $\mathcal{R}_{T_c}$ have been computed for all children T_c of T. We show how to compute $\mathcal{R}_T$ using an associated VASS. For each truth assignment β of Φ_T, we construct a VASS $\mathcal{V}(T, \beta) = (Q, A)$ as follows. The states in Q are all tuples $(\tau, \sigma, q, \bar{o}, \bar{c}_{ib})$ where τ is a T-isomorphism type, σ a service, q a state of $B(T, \beta)$, and $\bar{c}_{ib}$ a mapping from $\mathit{TS}_{ib}(T)$ to $\{0, 1\}$. The vector $\bar{o}$ indicates the current stage of each child T_c of T (`init`, `active` or `inactive`) and also specifies the outputs of T_c (an isomorphism type or $\perp$). That is, $\bar{o}$ is a partial mapping associating to some of the children T_c of T the value $\perp$, a T_c-isomorphism type projected to $\bar{x}^{T_c}_{\mathsf{in}} \cup \bar{x}^{T_c}_{\mathsf{out}}$ or the value `inactive`. Intuitively, $T_c \notin \mathit{dom}(\bar{o})$ means that T_c is in the `init` state, and $\bar{o}(T_c) = \perp$ indicates that T_c has been called but will not return. If $\bar{o}(T_c)$ is an isomorphism type τ, this indicates that T_c has been called, has not yet returned, and will return the isomorphism type τ. When T_c returns, $\bar{o}(T_c)$ is set to `inactive`, and T_c cannot be called again before an internal service of T is applied.

The set of actions A consists of all triples $(\alpha, \bar{a}, \alpha')$ where $\alpha = (\tau, \sigma, q, \bar{o}, \bar{c}_{ib})$, $\alpha' = (\tau', \sigma', q', \bar{o}', \bar{c}'_{ib})$, δ' is the update of σ', and the following hold:

- τ' is a successor of τ by applying service σ';
- $\bar{a} = \bar{a}(\delta', \hat{\tau}, \hat{\tau}', \bar{c}_{ib})$ (defined in Section 3.5.1), where $\hat{\tau} = \tau|(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$ and $\hat{\tau}' = \tau'|(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$
- $\bar{c}'_{ib} = \bar{c}_{ib} + \bar{a}$
- if σ' is an internal service, $\mathit{dom}(\bar{o}') = \emptyset$.
- If $\sigma' = \sigma^o_{T_c}$, then $T_c \notin \mathit{dom}(\bar{o})$ and for $\tau^{T_c}_{in} = f^{-1}_{in}(\tau|(\bar{x}^T_{T_c^{\downarrow}}, h(T_c)))$, for some output $\tau^{T_c}_{out}$ of T_c and truth assignment β^{T_c} to Φ_{T_c}, tuple $(\tau^{T_c}_{in}, \tau^{T_c}_{out}, \beta^{T_c})$ is in $\mathcal{R}_{T_c}$. Note that $\tau^{T_c}_{out}$ can be $\perp$, which indicates that this call to T_c does not return. Also, $\bar{o}' = \bar{o}[T_c \mapsto \tau^{T_c}_{out}]$.
- If $\sigma' = \sigma^c_{T_c}$, then $\bar{o}(T_c) = (f^{-1}_{out} \circ f^{-1}_{in})(\tau'|(\bar{x}^T_{T_c^{\downarrow}} \cup \bar{x}^T_{T_c^{\uparrow}}, h(T_c)))$ and $\bar{o}' = \bar{o}[T_c \mapsto \mathtt{inactive}]$.
- q' is a successor of q in $B(T, \beta)$ by evaluating Φ_T using (τ', σ'). If $\sigma' = \sigma^o_{T_c}$, formulas in Φ_{T_c} are assigned the truth values defined by β^{T_c}.

An *initial* state of $\mathcal{V}(T,\beta)$ is a state of the form $v_0 = (\tau_0, \sigma_0, q_0, \bar{o}_0, \bar{c}^0_{ib})$ where τ_0 is an initial T-isomorphism type (i.e., for every $x \in \bar{x}^T_{id} - \bar{x}^T_{in}$, $x \sim_{\tau_0}$ `null`, and for every $x \in \bar{x}^T_{\mathbb{R}} - \bar{x}^T_{in}$, $x \sim_{\tau_0} 0$), $\sigma_0 = \sigma^o_T$, q_0 is the successor of some initial state of $B(T,\beta)$ under (τ_0, σ_0), $dom(\bar{o}_0) = \emptyset$, and $\bar{c}^0_{ib} = \bar{0}$.

Computing $\mathcal{R}_T(\tau_{in}, \tau_{out}, \beta)$ from $\mathcal{V}(T,\beta)$. Checking whether $(\tau_{in}, \tau_{out}, \beta)$ is in $\mathcal{R}_T$ can be done using a (repeated) reachability test on $\mathcal{V}(T,\beta)$, as stated in the following key lemma.

Lemma 65. *$(\tau_{in}, \tau_{out}, \beta) \in \mathcal{R}_T$ iff there exists an initial state $v_0 = (\tau_0, \sigma_0, q_0, \bar{o}_0, \bar{c}^0_{ib})$ of $\mathcal{V}(T,\beta)$ for which $\tau_0|\bar{x}^T_{\mathsf{in}} = \tau_{in}$ and the following hold:*

- *If $\tau_{out} \neq \bot$, then there exists a state $v_n = (\tau_n, \sigma_n, q_n, \bar{o}_n, \bar{c}^n_{ib})$ where $\tau_{out} = \tau_n|(\bar{x}^T_{\mathsf{in}} \cup \bar{x}^T_{\mathsf{out}})$, $\sigma_n = \sigma^c_T$, $q_n \in Q^{fin}$ where Q^{fin} is the set of accepting states of $B(T,\beta)$ for finite runs, such that v_n is reachable from v_0. A path from $(v_0, \bar{0})$ to $(v_n, \bar{z}_n)$ is called a **returning path**.*
- *If $\tau_{out} = \bot$, then one of the following holds:*
  - *there exists a state $v_n = (\tau_n, \sigma_n, q_n, \bar{o}_n, \bar{c}^n_{ib})$ in which $q_n \in Q^{inf}$ where Q^{inf} is the set of accepting states of $B(T,\beta)$ for infinite runs, such that v_n is repeatedly reachable from v_0. A path $(v_0, \bar{0}) \ldots (v_n, \bar{z}_n) \ldots (v_n, \bar{z}'_n)$ where $\bar{z}_n \leq \bar{z}'_n$ is called a **lasso path**.*
  - *there exists state $v_n = (\tau_n, \sigma_n, q_n, \bar{o}_n, \bar{c}^n_{ib})$ in which $\bar{o}_n(T_c) = \bot$ for some child T_c of T and $q_n \in Q^{fin}$, such that v_n is reachable from v_0. The path from $(v_0, \bar{0})$ to $(v_n, \bar{z}_n)$ is called a **blocking path**.*

The proof of Lemma 65 is by induction on the task hierarchy $\mathcal{H}$.

Proof. **Base Case.** Consider $\mathcal{R}_T(\tau_{in}, \tau_{out}, \beta)$ where T is a leaf task. As T has no subtask, $dom(\bar{o})$ is always empty so $\bar{o}$ can be ignored. Note that, by definition, there can be no blocking path of $\mathcal{V}(T,\beta)$.

For the *if* part, consider $(\tau_{in}, \tau_{out}, \beta) \in \mathcal{R}_T$. Suppose first that $\tau_{out} \neq \bot$. By definition, there exists a finite local symbolic run $(\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$ accepted by $B(T,\beta)$, where $\gamma \in \mathbb{N}$ and $\sigma_{\gamma-1} = \sigma^c_T$. Consider an accepting computation $\{q_i\}_{0 \leq i < \gamma}$ of $B(T,\eta)$ on $\{(I_i, \sigma_i)\}_{0 \leq i < \gamma}$, such that $q_{\gamma-1} \in Q_{fin}$. We can construct a returning path $P = \{(p_i, \bar{z}_i)\}_{0 \le i < \gamma}$ of $\mathcal{V}(T, \beta)$ where for each state $p_i = (\tau_i, \sigma_i, q_i, \bar{o}_i, \bar{c}^i_{ib})$, (τ_i, σ_i, q_i) is obtained directly from $\{(I_i, \sigma_i)\}_{0 \le i < \gamma}$ and $\{q_i\}_{0 \le i < \gamma}$, $\bar{z}_i = \bar{c}_i$, and $\bar{c}^i_{ib}$ is the projection of $\bar{c}_i$ to input-bound TS-isomorphism types.

Now suppose $\tau_{out} = \perp$. By definition, and since T is a leaf task, there exists an infinite symbolic run $(\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \omega})$ accepted by $B(T, \beta)$. Consider the sequence $\{q_i\}_{0 \le i < \omega}$ of states in an accepting computation of $B(T, \eta)$ on $\{(I_i, \sigma_i)\}_{0 \le i < \omega}$. There must exist $q_f \in Q_{inf}$ such that for infinitely many i, $q_i = q_f$. So we can construct a path $P = \{(p_i, \bar{z}_i)\}_{0 \le i < \omega}$ of $\mathcal{V}(T, \beta)$ where for each state $p_i = (\tau_i, \sigma_i, q_i, \bar{o}_i, \bar{c}^i_{ib})$ is obtained in the same way as in the case where $\tau_{out} \neq \perp$. By the Dickson's lemma [53], there exists two distinct indices m and n such that $m < n$, $(\tau_m, \sigma_m, q_m, \bar{c}^m_{ib}) = (\tau_n, \sigma_n, q_n, \bar{c}^n_{ib})$, $q_m = q_n = q_f$ and $\bar{z}_m \le \bar{z}_n$. Thus, the sequence $(p_0, \bar{z}_0), \ldots, (p_m, \bar{z}_m), \ldots, (p_n, \bar{z}_n)$ is a lasso path of $\mathcal{V}(T, \beta)$.

For the *only-if* direction, if there exists a returning path in $\mathcal{V}(T, \beta)$, then by definition, τ_{in} and τ_{out} together with the sequence $\{(I_i, \sigma_i)\}_{0 \le i \le n}$ where each (I_i, σ_i) is obtained directly from $(p_i, \bar{z}_i)$ is a valid local symbolic run $\tilde{\rho}_T$. And $\tilde{\rho}_T$ is accepted by $B(T, \beta)$ since q_n is in Q^{fin}. If there exists a lasso path in $\mathcal{V}(T, \beta)$, then we can obtain a finite sequence $\{(I_i, \sigma_i)\}_{0 \le i \le n}$ similar to above. And we can construct $\{(I_i, \sigma_i)\}_{0 \le i < \omega}$ by repeating the subsequence from index $m + 1$ to index n infinitely many times. As $q_n = q_f \in Q^{inf}$, $(\tau_{in}, \perp, \{(I_i, \sigma_i)\}_{0 \le i < \omega})$ is an infinite local symbolic run accepted by $B(T, \beta)$, so $(\tau_{in}, \perp, \beta) \in \mathcal{R}_T$.

Induction. Consider a non-leaf task T, and suppose the statement is true for all its children tasks.

For the *if* part, suppose $(\tau_{in}, \tau_{out}, \beta) \in \mathcal{R}_T$. Then there exists an adorned symbolic tree of runs ***Sym***$_T$ with root $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$ accepted by $\mathcal{B}_{\bar{\beta}}$. We construct a path $P = \{(p_i, \bar{z}_i)\}_{0 \le i < \gamma}$ of $\mathcal{V}(T, \beta)$ as follows. The transitions in $\tilde{\rho}_T$ caused by internal services are treated as in the base case. Suppose that $\sigma_i = \sigma^o_{T_c}$ for some child T_c of T. Then there is an edge labeled (i, β^{T_c}) from $\tilde{\rho}_T$ to a symbolic tree of runs accepted by $\mathcal{B}_{\bar{\beta}^{T_c}}$, rooted at a run $\tilde{\rho}_{T_c}$ of T_c with input $\tau^{T_c}_{in}$ and output $\tau^{T_c}_{out}$. Thus, $(\tau^{T_c}_{in}, \tau^{T_c}_{out}, \beta^{T_c}) \in \mathcal{R}_{T_c}$ and $\mathcal{V}(T, \beta)$ can make the transition from $(p_{i-1}, \bar{z}_{i-1})$ to $(p_i, \bar{z}_i)$ as in its definition (including the updates to $\bar{o}$). If $\tau^{T_c}_{out} \neq \perp$ then there exists a minimum $j > i$ for which $\sigma_j = \sigma^c_{T_c}$ and once again $\mathcal{V}(T, \beta)$ can make the transition from $(p_{j-1}, \bar{z}_{j-1})$ to $(p_j, \bar{z}_j)$ as in its definition, mimicking the return of T_c using the isomorphism type $\tau^{T_c}_{out}$ stored in $\bar{o}(T_c)$. Now consider the resulting path $P = \{(p_i, \bar{z}_i)\}_{0 \leq i < \gamma}$. By applying a similar analysis as in the base case, if $\gamma \neq \omega$ and $\tau_{out} \neq \perp$, then P is a returning path. If $\gamma \neq \omega$ and $\tau_{out} = \perp$, then P is a blocking path. If $\gamma = \omega$, then there exists a prefix P' of P such that P' is a lasso path.

For the *only-if* direction, let P be a path of $\mathcal{V}(T, \beta)$, starting from a state $p_0 = (\tau_0, \sigma_0, q_0, \bar{o}_0, \bar{c}^0_{ib})$ where $\tau_0 | \bar{x}^T_{\mathsf{in}} = \tau_{in}$. If P is a returning path, let $v_n = (\tau_n, \sigma_n, q_n, \bar{o}_n, \bar{c}^n_{ib})$ be its last state and $\tau_{out} = \tau_n | (\bar{x}^T_{\mathsf{in}} \cup \bar{x}^T_{\mathsf{out}})$. If P is not a returning path, then $\tau_{out} = \perp$. From P we can construct a adorned symbolic tree of runs $\textit{\textbf{Sym}}_T$ accepted by $\mathcal{B}_{\bar{\beta}}$ as follows. The root of $\textit{\textbf{Sym}}_T$ is a local symbolic run $\tilde{\rho}_T$ constructed analogously to the construction in the only-if direction in the base case. Then for each $\sigma_i = \sigma^o_{T_c}$, by the induction hypothesis, there exists a symbolic tree of runs $\textit{\textbf{Sym}}_{T_c}$ whose root has input isomorphism type $\tau^{T_c}_{in}$, output isomorphism type $\tau^{T_c}_{out}$ and is accepted by $\mathcal{B}_{\beta^{T_c}}$ (note that $\tau^{T_c}_{in}$, $\tau^{T_c}_{out}$ and β^{T_c} are uniquely defined by P and i). We connect $\textit{\textbf{Sym}}_T$ with $\textit{\textbf{Sym}}_{T_c}$ with an edge labeled (i, β^{T_c}).

If P is a returning or blocking path, then $\textit{\textbf{Sym}}_T$ is accepted by $\mathcal{B}_{\bar{\beta}}$. If P is a lasso path, then we first modify the root $\tilde{\rho}_T$ of $\textit{\textbf{Sym}}_T$ by repeating the subsequence from $m+1$ to n infinitely, then for each integer i such that $m + 1 \leq i \leq n$ and $\textit{\textbf{Sym}}_T$ is connected with some $\textit{\textbf{Sym}}_{T_c}$ with edge labeled index (i, β^{T_c}), for each repetition $I_{i'}$ of symbolic instance I_i, we make a copy of $\textit{\textbf{Sym}}_{T_c}$ and connect $\textit{\textbf{Sym}}_T$ with $\textit{\textbf{Sym}}_{T_c}$ with edge labeled (i', β^{T_c}). The resulting $\textit{\textbf{Sym}}_T$ is accepted by $\mathcal{B}_{\bar{\beta}}$. Thus, $(\tau_{in}, \tau_{out}, \beta) \in \mathcal{R}_T$. □

Complexity of verification. We now have all ingredients in place for our verification algorithm. Let Γ be a HAS and φ_f an HLTL-FO formula over Γ. In view of the previous development, $\Gamma \models \varphi_f$ iff $\neg\varphi_f$ is **not** satisfiable by a symbolic tree of runs of Γ. We outline a non-deterministic algorithm for checking satisfiability of $\neg\varphi_f$, and establish its space complexity $O(f)$, where f is a function of the relevant parameters. The space complexity of verification (the complement) is then upper bounded by $O(f^2)$ by Savitch's theorem [115].

Recall that φ_f is satisfiable by a symbolic tree of runs of Γ iff $(\tau_{in}, \perp, \beta) \in \mathcal{R}_{T_1}$ for some τ_{in} satisfying the precondition of Γ, and $\beta(\neg\varphi_f) = 1$. By Lemma 65, membership in $\mathcal{R}_{T_1}$ can be reduced to state (repeated) reachability in the VASS $\mathcal{V}(T_1, \beta)$. For a given VASS, (repeated) reachability is decided by non-deterministically generating runs of the VASS up to a certain length, using space $O(\log n \cdot 2^{c \cdot d \log d})$ where n is the number of states, d is the vector dimension and c is a constant [69]. The same approach can be used for the VASS $\mathcal{V}(T_1, \beta)$, with the added complication that generating transitions requires membership tests in the relations $\mathcal{R}_{T_c}$'s for $T_c \in child(T_1)$. These in turn become (repeated) reachability tests in the corresponding VASS. Assuming that n and d are upper bounds for the number of states and dimensions for all $\mathcal{V}(T, \beta)$ with $T \in \mathcal{H}$, this yields a total space bound of $O(h \log n \cdot 2^{c \cdot d \log d})$ for membership testing in $\mathcal{V}(T_1, \beta)$, where h is the depth of $\mathcal{H}$.

In our construction of $\mathcal{V}(T, \beta)$, the vector dimension d is the number of TS-isomorphism types. The number of states n is at most the product of the number of T-isomorphism types, the number states in $B(T, \beta)$, the number of all possible $\bar{o}$ and the number of possible states of $\bar{c}_{ib}$. The worst-case complexity occurs for HAS with unrestricted schemas (cyclic foreign keys) and artifact relations. To understand the impact of the foreign key structure and artifact relations, we also consider the complexity for acyclic and linear-cyclic schemas, and without artifact relations. A careful analysis yields the following (see Appendix 3.10). For better readability, we state the complexity for HAS over a fixed schema (database and maximum arity of artifact relations). The impact of the schema is detailed in Appendix 3.10.

Theorem 66. *Let Γ be a HAS over a fixed schema and φ_f an HLTL-FO formula over Γ. The deterministic space complexity upper bounds of checking whether $\Gamma \models \varphi_f$ are summarized in Table 3.1.* [5]

Note that the worst-case space complexity is non-elementary, as for feedback-free systems [36]. However, the height of the tower of exponentials in [36] is the square of the total number

[5] k- exp is the tower of exponential functions of height k.

Table 3.1. Space complexity upper bounds for verification without arithmetic (N: size of (Γ, φ_f); h: depth of hierarchy; c: constants depending on the schema).

	Acyclic	Linearly-Cyclic	Cyclic
w/o. Artifact relations	$c \cdot N^{O(1)}$	$O(N^{c \cdot h})$	$h\text{-}\exp(O(N))$
w. Artifact relations	$O(\exp(N^c))$	$O(2\text{-}\exp(N^{c \cdot h}))$	$(h+2)\text{-}\exp(O(N))$

of artifact variables of the system, whereas in our case it is the depth of the hierarchy, likely to be much smaller.

Lower bounds. Several complexity lower bounds for verification can be immediately obtained. When there are no artifact relations, the verification problem is PSPACE-hard, as LTL model checking alone is already PSPACE-complete [124]. When artifact relations are present, one can show EXPSPACE-hardness by a direct simulation of VASS by HAS, and the fact that model checking for VASS is EXPSPACE-complete [91, 107, 69]. These lower bounds are tight when the schema is acyclic. Lower bounds for the other cases remain open.

3.6 Verification with Arithmetic

We next outline the extension of our verification algorithm to handle HAS and HLTL-FO properties whose conditions use arithmetic constraints expressed as polynomial inequalities with integer coefficients over the numeric variables (ranging over $\mathbb{R}$). We note that one could alternatively limit the arithmetic constraints to linear inequalities with integer coefficients (and variables ranging over $\mathbb{Q}$), yielding the same complexity. These are sufficient for many applications.

The seed idea behind our approach is that, in order to determine whether the arithmetic constraints are satisfied, we do not need to keep track of actual valuations of the task variables and the numeric navigation expressions they anchor (for which the search space would be infinite). Instead, we show that these valuations can be partitioned into a finite set of equivalence classes with respect to satisfaction of the arithmetic constraints, which we then incorporate into the isomorphism types of Section 3.5, extending the algorithm presented there. This however raises some significant technical challenges, which we discuss next.

Intuitively, this approach uses the fact that a finite set of polynomials $\mathcal{P}$ partitions the space into a bounded number of *cells* containing points located in the same region ($= 0, < 0, > 0$) with respect to every polynomial $P \in \mathcal{P}$. Isomorphism types are extended to include a cell, which determines which arithmetic constraints are satisfied in the conditions of services and in the property. In addition to the requirements detailed in Section 3.5, we need to enforce cell compatibility across symbolic service calls. For instance, when a task executes an internal service, the corresponding symbolic transition from cell c to c' is possible only if the projections of c and c' on the subspace corresponding to the task's input variables have non-empty intersection (since input variables are preserved). Similarly, when the opening or closing service of a child task is called, compatibility is required between the parent's and the child's cell on the shared variables, which amounts again to non-empty intersection between cell projections. This suggests the following first-cut (and problematic) attempt at a verification algorithm: once a local transition imposes new constraints, represented by a cell c', these constraints are propagated *back* to previously guessed cells, refining them via intersection with c'. If an intersection becomes empty, the candidate symbolic run constructed so far has no corresponding actual run and the search is pruned. The problem with this attempt is that it is incompatible with the way we deal with sets in Section 3.5: the contents of sets are represented by associating counters to the isomorphism types of their elements. Since extended isomorphism types include cells, retroactive cell intersection invalidates the counters and the results of previous VASS reachability checks.

We develop an alternative solution that avoids retroactive cell intersection altogether. More specifically, for each task, our algorithm extends isomorphism types with cells guessed from a *pre-computed* set constructed by following the task hierarchy bottom-up and including in the parent's set those cells obtained by appropriately projecting the children's cells on shared variables and expressions. Only non-empty cells are retained. We call the resulting cell collection the Hierarchical Cell Decomposition (HCD).

The key benefit of the HCD is that it arranges the space of cells so that consistency of a symbolic run can be guaranteed by performing simple local compatibility tests on the cells involved in each transition. Specifically, (i) in the case of internal service calls, the next cell c' must *refine* the current cell c on the shared variables (that is, the projection of c' must be contained in the projection of c); (ii) in the case of child task opening/closing services, the parent cell c must refine the child cell c'. This ensures that in case (i) the intersection with c' of all relevant previously guessed cells is non-empty (because we only guess non-empty cells and c' refines all prior guesses), and in case (ii) the intersection with the child's cell c' is a no-op for the parent cell. Consequently, retroactive intersection can be skipped as it can never lead to empty cells.

A natural starting point for constructing the HCD is to gather for each task all the polynomials appearing in its arithmetic constraints (or in the property sub-formulas referring to that task), and associate sign conditions to each. This turns out to be insufficient. For example, the projection from the child cell can impose on the parent variables new constraints which do not appear explicitly in the parent task. It is a priori not obvious that the constrained cells can be represented symbolically, let alone efficiently computed. The tool enabling our solution is the Tarski-Seidenberg Theorem [122], which ensures that the projection of a cell is representable by a union of cells defined by a set of polynomials (computed from the original ones) and sign conditions for them. The polynomials can be efficiently computed using quantifier elimination.

Observe that a bound on the number of newly constructed polynomials yields a bound on the number of cells in the HCD, which in turn implies a bound on the number of distinct extended isomorphism types manipulated by the verification algorithm, ultimately yielding decidability of verification. A naive analysis produces a bound on the number of cells that is hyperexponential in the height of the task hierarchy, because the number of polynomials can proliferate at this rate when constructing all possible projections, and p polynomials may produce 3^p cells. Fortunately, a classical result from real algebraic geometry ([8], reviewed in Section 3.6.2) bounds the number of distinct *non-empty* cells to only exponential in the number of variables (the exponent is independent of the number of polynomials). This yields an upper bound of the number of cells (and also the number of extended isomorphism types) which is singly exponential in the number of numeric expressions and doubly exponential in the height of the hierarchy $\mathcal{H}$. We state below our complexity results for verification with arithmetic, relegating details (including a fine-grained analysis) to the remainder of this section.

Theorem 67. *Let Γ be a HAS over a fixed database schema and φ_f an HLTL-FO formula over Γ. If arithmetic is allowed in (Γ, φ_f), then the deterministic space complexity uuper bounds for checking whether $\Gamma \models \varphi_f$ are summarized in Table 3.2.*

Table 3.2. Space complexity upper bounds for verification with arithmetic (N: size of (Γ, φ); h: depth of hierarchy; c: constants depending on the schema.)

	Acyclic	Linearly-Cyclic	Cyclic
w/o. Artifact relations	$O(\exp(N^{c \cdot h}))$	$O(\exp(N^{c \cdot h^2}))$	$(h+1)\text{-}\exp(O(N))$
w. Artifact relations	$O(2\text{-}\exp(N^{c \cdot h}))$	$O(2\text{-}\exp(N^{c \cdot h^2}))$	$(h+2)\text{-}\exp(O(N)))$

3.6.1 Background on Quantifier Elimination

We next outline the technical details for verification with arithmetic, starting with a review of quantifier elimination and real algebraic geometry. The quantifier elimination (QE) problem for the reals can be stated as follows.

Definition 68. *For real variables $Y = \{y_i\}_{1 \leq i \leq l}$ and a formula $\Phi(Y)$ of the form $(Q_1 x_1) \dots (Q_k x_k)$ $F(y_1 \dots y_l, x_1 \dots x_k)$ where $Q_i \in \{\exists, \forall\}$ and $F(y_1 \dots y_l, x_1 \dots x_k)$ is a Boolean combination of polynomial inequalities with integer coefficients, the quantifier elimination problem is to output a quantifier-free formula $\Psi(Y)$ such that for every $Y \in \mathbb{R}^l$, $\Phi(Y)$ is true iff $\Psi(Y)$ is true.*

The best known algorithm for solving the QE problem for the reals has time and space complexity doubly-exponential in the number of quantifier alternations and singly-exponential in the number of variables. When applying QE in verification of HAS, we are only interested in formulas that are existentially quantified. According to Algorithm 14.6 of [7], the result for this special case can be stated as follows:

Theorem 69. *For existentially quantified formula $\Phi(Y)$, an equivalent quantifier-free formula $\Psi(Y)$ can be computed in time and space $(s \cdot d)^{O(k)O(l)}$, where s is the number of polynomials in Φ, d is the maximum degree of the polynomials, k is the quantifier rank of Φ and $l = |Y|$.*

Note that in the special case when $l = 0$, quantifier elimination simply checks satisfiability. Thus we have:

Corollary 70. *Satisfiability over the reals of a Boolean combination Φ of polynomial inequalities with integer coefficients can be decided in time and space $(s \cdot d)^{O(k)}$, where s is the number of polynomials in Φ, d is the maximum degree of the polynomials, and k is the number of variables in Φ.*

Also in Section 14.3 of [7], it is shown that if the bit-size of coefficients in Φ is bounded by τ, then the bit-size of coefficients in Ψ is bounded by $\tau \cdot d^{O(k)O(l)}$.

3.6.2 Background on General Real Algebraic Geometry

We next review a classic result in general real algebraic geometry.

Definition 71. *For a given set of polynomials $\mathcal{P} = \{P_1, \ldots, P_s\}$ over k variables $\{x_i\}_{1 \le i \le k}$, a sign condition of $\mathcal{P}$ is a mapping $\sigma : \mathcal{P} \mapsto \{-1, 0, +1\}$. We denote by $\kappa(\sigma, \mathcal{P})$ the semialgebraic set $\{x | x \in \mathbb{R}^k, sign(P(x)) = \sigma(P), \forall P \in \mathcal{P}\}$[6] called the* cell *of the sign condition σ for $\mathcal{P}$.*

We use the following result from [72, 8]:

Theorem 72. *Given a set of polynomials $\mathcal{P}$ with integer coefficients over k variables $\{x_i\}_{1 \le i \le k}$, the number of distinct non-empty cells, namely $\#\{\sigma : \mathcal{P} \mapsto \{-1, 0, +1\} \, | \kappa(\sigma, \mathcal{P}) \neq \emptyset\}$, is at most $(s \cdot d)^{O(k)}$, where $s = |\mathcal{P}|$ and d is the maximum degree of polynomials in $\mathcal{P}$.*

[6]The sign function $sign(x)$ equals -1 if $x < 0$, 0 if $x = 0$ and 1 if $x > 0$.

Given a set of polynomials $\mathcal{P}$, we can use the following naive approach to compute the set of sign conditions resulting in non-empty cells. We simply enumerate sign conditions of $\mathcal{P}$ and discard sign conditions that results in empty cells or cells equivalent to any recorded sign conditions known to be non-empty. Checking whether a cell is empty and checking whether two cells are equivalent can be reduced to checking satisfiability of a formula of polynomial inequalities. By Corollary 70, this naive approach takes space $(s \cdot d)^{O(k)}$.

Theorem 73. *Given a set of polynomials $\mathcal{P}$ over $\{x_i\}_{1 \leq i \leq k}$, the set of non-empty cells $\{\sigma : \mathcal{P} \mapsto \{-1, 0, +1\} \mid \kappa(\sigma, \mathcal{P}) \neq \emptyset\}$ defined by $\mathcal{P}$ can be computed in space $(s \cdot d)^{O(k)}$ where $s = |\mathcal{P}|$ and d is the maximum degree of polynomials in $\mathcal{P}$.*

3.6.3 Cells for Verification

Intuitively, in order to handle arithmetic in our verification framework, we need to extend each isomorphism type τ with a set of polynomial inequality constraints over the set of numeric expressions in the extended navigation set $\mathcal{E}_T^+$.

We say that an expression e is numeric if $e = x$ for some numeric variable x or $e = x_R.w$ and the last attribute of w is numeric. For each task T, we denote by $\mathcal{E}_{\mathbb{R}}^T$ the set of numeric expressions of T where for each $x_R.w \in \mathcal{E}_{\mathbb{R}}^T$, $|w| \leq h(T)$.

The constraints over the numeric expressions are represented by a non-empty cell κ (formally defined below). When a service is applied, the arithmetic parts of the conditions are evaluated against κ. And for every transition $I \xrightarrow{\sigma'} I'$ where κ, κ' are the cells of I, I' respectively, if any variables are modified by the transition, then the projection of κ' onto the preserved numeric expressions has to *refine* the projection of κ onto the preserved numeric expressions. Similar compatibility checks are required when a child task returns to its parent.

We introduce some more notation. For every $T \in \mathcal{H}$, we consider polynomials in the polynomial ring $\mathbb{Z}[\mathcal{E}_{\mathbb{R}}^T]$. For each polynomial P, we denote by $var(P)$ the set of numeric expressions mentioned in P and for a set of polynomials $\mathcal{P}$, we denote by $var(\mathcal{P})$ the set

$\bigcup_{P \in \mathcal{P}} var(P)$. For $\mathcal{P} \subset \mathbb{Z}[\mathcal{E}^T_{\mathbb{R}}]$ and $\mathcal{E} \subseteq \mathcal{E}^T_{\mathbb{R}}$, we denote by $\mathcal{P}|\mathcal{E}$ the set of polynomials $\{P|P \in \mathcal{P}, var(P) \subseteq \mathcal{E}\}$.

We next define the cells used in our verification algorithm. At task T, for a set of numeric expressions $\mathcal{E} \subseteq \mathcal{E}^T_{\mathbb{R}}$ and a set of polynomials $\mathcal{P}$ where $var(\mathcal{P}) \subseteq \mathcal{E}$, we define the cells over $(\mathcal{E}, \mathcal{P})$ as follows.

Definition 74. *A cell κ over $(\mathcal{E}, \mathcal{P})$ is a subset of $\mathbb{R}^{|\mathcal{E}|}$ for which there exists a sign condition σ of $\mathcal{P}$ such that $\kappa = \kappa(\sigma, \mathcal{P})$.*

For $\mathcal{P} \subset \mathbb{Z}[\mathcal{E}^T_{\mathbb{R}}]$, we denote by $\mathcal{K}(\mathcal{P}, \mathcal{E})$ the set of cells over $(\mathcal{E}, \mathcal{P}|\mathcal{E})$. Namely, $\mathcal{K}(\mathcal{P}, \mathcal{E}) = \{\kappa(\sigma, \mathcal{P}|\mathcal{E}) | \sigma \in \mathcal{P}|\mathcal{E} \mapsto \{-1, 0, +1\}\}$. And we denote by $\mathcal{K}(\mathcal{P})$ the set of cells $\bigcup_{\mathcal{E} \subseteq \mathcal{E}^T_{\mathbb{R}}} \mathcal{K}(\mathcal{P}, \mathcal{E})$.

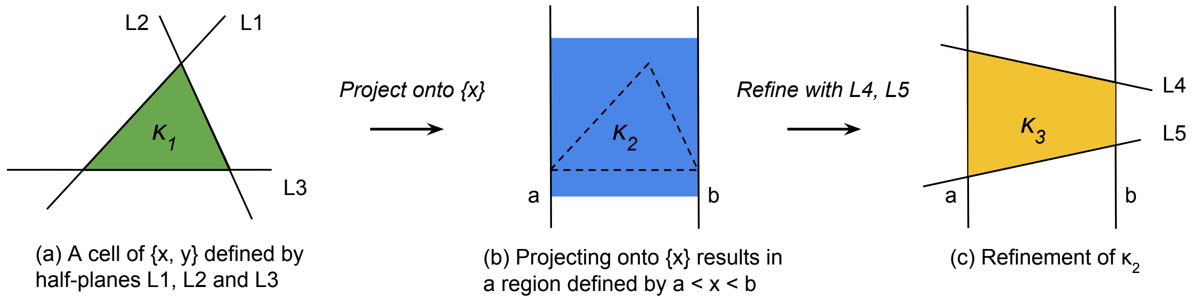
Compatibility between cells is tested using the notion of refinement. Intuitively, a cell κ refines another cell κ' if κ can be obtained by adding extra numeric expressions and/or constraints to κ'. Formally,

Definition 75. *For cell κ over $(\mathcal{E}, \mathcal{P})$ and cell κ' over $(\mathcal{E}', \mathcal{P}')$ where $\kappa = \kappa(\sigma, \mathcal{P})$ and $\kappa' = \kappa(\sigma', \mathcal{P}')$, we say that κ refines κ', denoted by $\kappa \sqsubseteq \kappa'$, if $\mathcal{E}' \subseteq \mathcal{E}$, $\mathcal{P}' \subseteq \mathcal{P}$ and $\sigma|\mathcal{P}' = \sigma'$. Note that if $\mathcal{E} = \mathcal{E}'$, then $\kappa \sqsubseteq \kappa'$ iff $\kappa \subseteq \kappa'$.*

We next define the projection of a cell onto a set of variables. For each cell κ over $(\mathcal{E}, \mathcal{P})$ where $\mathcal{E} \subseteq \mathcal{E}^T_{\mathbb{R}}$ and variables $\bar{x} \subseteq \bar{x}^T$, the projection of κ onto $\bar{x}$, denoted by $\kappa|\bar{x}$, is defined to be the projection of κ onto the expressions $\mathcal{E}|\bar{x}$ where $\mathcal{E}|\bar{x} = \{e \in \mathcal{E} | e = x_R.w \vee e = x, x \in \bar{x}\}$. By the Tarski-Seidenberg theorem [122], $\kappa|\bar{x}$ is a union of disjoint cells. Also, the projections $\kappa|\bar{x}$ can be obtained by quantifier elimination. Let $\Phi(\kappa)$ be the conjunctive formula defining κ using polynomials in $\mathcal{P}$. Then by treating $\mathcal{E}|\bar{x}$ as the set of free variables, the formula $\Psi(\kappa)$ obtained by eliminating $\mathcal{E} - \mathcal{E}|\bar{x}$ from $\Phi(\kappa)$ defines $\kappa|\bar{x}$. We denote by $\texttt{proj}(\kappa, \bar{x})$ the set of polynomials mentioned in $\Psi(\kappa)$. It is easy to see that $\kappa|\bar{x}$ is a union of cells over $(\mathcal{E}|\bar{x}, \texttt{proj}(\kappa, \bar{x}))$.

The following notation is useful for checking compatibility between a cell and the projection of another cell: we define that a cell κ refines another cell κ' wrt to projection to $\bar{x}$, denoted as $\kappa \sqsubseteq_{\bar{x}} \kappa'$, if there exists a cell $\tilde{\kappa} \subseteq \kappa'|\bar{x}$ such that $\kappa \sqsubseteq \tilde{\kappa}$.

Example 76. *We illustrate the key notations of cells, projection and refinement in Figure 3.14. Consider a task T with only two numeric variables $\{x, y\}$ where x is an input variable. Figure 3.14(a) illustrates a cell κ_1 defined by 3 half-planes L_1, L_2 and L_3, which are polynomial constraints from the task T. When an internal service of T is applied, the pre-condition is first evaluated against $\{L_1, L_2, L_3\}$, then the value of x is propagated, so the triangular region κ_1 is projected to the x-axis and comes the cell κ_2 illustrated in Figure 3.14(b). Finally, the post-condition is evaluated thus the resulting cell κ_3 is a refinement of κ_2 with the additional constraints L_4 and L_5 from the post-condition.*



L2
L1
K1
L3
(a) A cell of {x, y} defined by half-planes L1, L2 and L3
Project onto {x}
K2
a
b
(b) Projecting onto {x} results in a region defined by a < x < b
Refine with L4, L5
K3
L4
L5
a
b
(c) Refinement of κ2

Figure 3.14. Illustration of cells, projection, and refinement

Finally, we introduce notation relative to variable passing between parent task and child task. For each task T and $T_c \in child(T)$, we denote by $\mathcal{E}_{\mathbb{R}}^{T_c \to T}$ the set of numeric expressions $\{e | e \in \mathcal{E}_{\mathbb{R}}^{T_c}, e = x \vee e = x_R.w, x \in \bar{x}_{\text{in}}^{T_c} \cup \bar{x}_{\text{out}}^{T_c}\}$. In other words, $\mathcal{E}_{\mathbb{R}}^{T_c \to T}$ is the subset of expressions in $\mathcal{E}_{\mathbb{R}}^{T_c}$ connected with expressions in $\mathcal{E}_{\mathbb{R}}^{T}$ by calls/returns of T_c. Let f_{in}, f_{out} be the input and output mapping between T and T_c. For each expression $e \in \mathcal{E}_{\mathbb{R}}^{T_c \to T}$, we define $e^{T_c \to T}$ to be an expression in $\mathcal{E}_{\mathbb{R}}^{T}$ as follows. If $e = x$, then $e^{T_c \to T} = (f_{in} \circ f_{out})(x)$. If $e = x_R.w$, then $e^{T_c \to T} = ((f_{in} \circ f_{out})(x))_R.w$. For a set of variables $\mathcal{E} \subseteq \mathcal{E}_{\mathbb{R}}^{T_c \to T}$, we define $\mathcal{E}^{T_c \to T}$ to be $\{e^{T_c \to T} | e \in \mathcal{E}\}$. For a polynomial P over $\mathcal{E}_{\mathbb{R}}^{T_c \to T}$ where $T_c \in child(T)$, we denote by $P^{T_c \to T}$ the polynomial obtained by replacing in P each numeric expression e with $e^{T_c \to T}$. For a cell κ of T_c where $\kappa = \kappa(\sigma, \mathcal{P})$ and $var(P) \subseteq \mathcal{E}_{\mathbb{R}}^{T_c \to T}$ for every $P \in \mathcal{P}$, we let $\kappa^{T_c \to T}$ to be the cell of T which equals $\kappa(\sigma', \mathcal{P}')$, where $\mathcal{P}' = \{P^{T_c \to T} | P \in \mathcal{P}\}$ and σ' is a sign condition over $\mathcal{P}'$ such that $\sigma'(P^{T_c \to T}) = \sigma(P)$ for every $P \in \mathcal{P}$.

3.6.4 Hierarchical Cell Decomposition

We now introduce the Hierarchical Cell Decomposition. Intuitively, for each task T, we would like to compute a set of polynomials $\mathcal{P}$ and a set of cells $\mathcal{K}_T$ such that for each subset $\mathcal{E}$ of $\mathcal{E}_{\mathbb{R}}^T$, the set of cells over $(\mathcal{E}, \mathcal{P}|\mathcal{E})$ in $\mathcal{K}_T$ is a partition of $\mathbb{R}^{|\mathcal{E}|}$.

The set of cells $\mathcal{K}_T$ satisfies the property that for the set of polynomials $\mathcal{P}$ mentioned at any condition of T in the specification Γ and HLTL-FO property φ_f, each cell $\kappa \in \mathcal{K}_T$ uniquely defines the sign condition of $\mathcal{P}$. This allows us to compute the signs of any polynomial in any condition in the local symbolic runs. In addition, for each pair of cells $\kappa, \kappa' \in \mathcal{K}_T$, we require that the projection of κ and κ' to the input variables $\bar{x}_{\mathsf{in}}^T$ (and $\bar{x}_{\mathsf{in}}^T \cup \bar{s}^T$) be disjoint or identical. So to check whether two cells κ and κ' of two consecutive symbolic instances in a local symbolic run are compatible when applying an internal service, we simply need to check whether their projections on $\bar{x}_{\mathsf{in}}^T$ are equal (note that refinement is implied by equality). Finally, for each child task T_c of T, for each cell $\kappa \in \mathcal{K}_T$ and $\kappa' \in \mathcal{K}_{T_c}$, κ uniquely defines the sign condition for the set of polynomials that defines $\kappa'|\bar{x}_{\mathsf{in}}^{T_c}$ and $\kappa'|(\bar{x}_{\mathsf{out}}^{T_c} \cup \bar{x}_{\mathsf{in}}^{T_c})$. This reduces to the problem of checking cell refinements when child tasks are called or return.

The Hierarchical Cell Decomposition is formally defined as follows.

Definition 77. *The Hierarchical Cell Decomposition associated to an artifact system $\mathcal{H}$ and property φ_f is a collection $\{\mathcal{K}_T\}_{T \in \mathcal{H}}$ of sets of cells, such that for each $T \in \mathcal{H}$, $\mathcal{K}_T = \mathcal{K}(\mathcal{P}'_T)$, where the set of polynomials $\mathcal{P}'_T$ is defined as follows. First, let $\mathcal{P}_T$ consist of the following:*

(1) all polynomials mentioned in any condition over $\bar{x}^T$ in Γ and the property φ_f,

(2) polynomials $\{e | e \in \mathcal{E}_{\mathbb{R}}^T\} \cup \{e - e' | e, e' \in \mathcal{E}_{\mathbb{R}}^T\}$, and

(3) for every $T_c \in child(T)$ and subset $\bar{x} \subseteq \bar{x}_{\mathsf{out}}^{T_c}$, the set of polynomials $\{P^{T_c \to T} | P \in \mathtt{proj}(\kappa, \bar{x}_{\mathsf{in}}^{T_c} \cup \bar{x}), \kappa \in \mathcal{K}_{T_c}\}$.

Next, let $\mathcal{P}_T^s = \mathcal{P}_T \cup \bigcup_{c \in \mathcal{K}(\mathcal{P}_T)} \mathtt{proj}(\kappa, \bar{x}_{\mathsf{in}}^T \cup \bar{s}^T)$. Finally, $\mathcal{P}'_T = \mathcal{P}_T^s \cup \bigcup_{c \in \mathcal{K}(\mathcal{P}_T^s)} \mathtt{proj}(\kappa, \bar{x}_{\mathsf{in}}^T)$.

Intuitively, when T is a leaf task, the set of cells $\mathcal{K}_T$ is constructed simply from (1) all polynomials within services of T and the parts of HLTL-FO property φ_f related T and (2) all possible tests of equality. When T is a non-leaf task, in addition to (1) and (2), we also need to take into account the constraints propagated to T from each child task T_c of T, which can be obtained by projecting cells in $\mathcal{K}_{T_c}$ onto T_c's input/output variables, resulting in the set of polynomials in (3).

The Hierarchical Cell Decomposition satisfies the following property, as desired.

Lemma 78. *Let T be a task and $\mathcal{P}'_T$ as above. For every pair of cells $\kappa_1, \kappa_2 \in \mathcal{K}_T$, and $\bar{x} = (\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$ or $\bar{x} = \bar{x}^T_{\mathsf{in}}$, if $\kappa_1 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E}_1)$ and $\kappa_2 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E}_2)$ where $\mathcal{E}_1|\bar{x} = \mathcal{E}_2|\bar{x}$, then $\kappa_1|\bar{x}$ and $\kappa_2|\bar{x}$ are either equal or disjoint.*

Proof. We prove the lemma for the case when $\bar{x} = \bar{x}^T_{\mathsf{in}}$. The proof is similar for $\bar{x} = \bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$.

Let $\tilde{\mathcal{P}}^s_T = \bigcup_{c \in \mathcal{K}(\mathcal{P}^s_T)} \texttt{proj}(\kappa, \bar{x}^T_{\mathsf{in}})$. For each cell $\kappa \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E})$, since $\mathcal{P}'_T|\mathcal{E} = (\mathcal{P}^s_T|\mathcal{E}) \cup (\tilde{\mathcal{P}}^s_T|\mathcal{E})$ as $\mathcal{P}'_T = \mathcal{P}^s_T \cup \tilde{\mathcal{P}}^s_T$, there exist $\kappa_1 \in \mathcal{K}(\mathcal{P}^s_T, \mathcal{E})$ and $\kappa_2 \in \mathcal{K}(\tilde{\mathcal{P}}^s_T, \mathcal{E})$ such that $\kappa = \kappa_1 \cap \kappa_2$. Then consider $\kappa|\bar{x}^T_{\mathsf{in}}$. Since all polynomials in $\tilde{\mathcal{P}}^s_T$ are over expressions of $\bar{x}^T_{\mathsf{in}}$, we have $\kappa|\bar{x}^T_{\mathsf{in}} = (\kappa_1 \cap \kappa_2)|\bar{x}^T_{\mathsf{in}} = (\kappa_1|\bar{x}^T_{\mathsf{in}}) \cap \kappa_2$. And by definition, $\texttt{proj}(\kappa_1, \bar{x}^T_{\mathsf{in}}) \subseteq \tilde{\mathcal{P}}^s_T$, so κ_2 uniquely defines the sign conditions for $\texttt{proj}(\kappa_1, \bar{x}^T_{\mathsf{in}})$, which means that either $\kappa_2 \cap \kappa_1|\bar{x}^T_{\mathsf{in}} = \emptyset$ or $\kappa_2 \subseteq \kappa_1|\bar{x}^T_{\mathsf{in}}$. And as $\kappa_2 \cap \kappa_1|\bar{x}^T_{\mathsf{in}} = \kappa|\bar{x}^T_{\mathsf{in}}$ is non-empty, $\kappa|\bar{x}^T_{\mathsf{in}} = \kappa_2$.

Therefore, for every $\kappa_1 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E}_1)$ and $\kappa_2 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E}_2)$ where $\mathcal{E}_1|\bar{x}^T_{\mathsf{in}} = \mathcal{E}_2|\bar{x}^T_{\mathsf{in}} = \mathcal{E}$, there exist cells $\tilde{\kappa}_1, \tilde{\kappa}_2 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E})$ such that $\kappa_1|\bar{x}^T_{\mathsf{in}} = \tilde{\kappa}_1$ and $\kappa_2|\bar{x}^T_{\mathsf{in}} = \tilde{\kappa}_2$. Since $\tilde{\kappa}_1$ and $\tilde{\kappa}_2$ are either disjoint or equal, $\kappa_1|\bar{x}^T_{\mathsf{in}}$ and $\kappa_2|\bar{x}^T_{\mathsf{in}}$ are also either disjoint or equal. □

From the above lemma, the following Corollary is obvious:

Corollary 79. *For every task T and $\kappa \in \mathcal{K}_T$, $\kappa|\bar{x}^T_{\mathsf{in}}$ and $\kappa|(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$ are single cells in $\mathcal{K}_T$.*

In view of the corollary, we use the notations of single-cell operators (projection, refinement, etc.) on $\kappa|\bar{x}^T_{\mathsf{in}}$ and $\kappa|(\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$ in the rest of our discussion.

To be able to connect with child tasks, we show the following property of $\mathcal{K}_T$:

Lemma 80. *For all tasks T and T_c where $T_c \in child(T)$, and every cell $\kappa_1 \in \mathcal{K}_T$ and $\kappa_2 \in \mathcal{K}_{T_c}$ where $\kappa_1 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E}_1)$ and $\kappa_2 \in \mathcal{K}(\mathcal{P}'_{T_c}, \mathcal{E}_2)$, for each set of variables $\bar{x} = \bar{x}^T_{T_c^\uparrow} \cup \bar{y}$ where $\bar{y}$ is some subset of $\bar{x}^T_{T_c^\downarrow}$, if $\mathcal{E}_1|\bar{x} = (\mathcal{E}_2)^{T_c \to T}|\bar{x}$, then either (1) $\kappa_1 \sqsubseteq_{\bar{x}} (\kappa_2)^{T_c \to T}$ or (2) $\kappa_1|\bar{x}$ is disjoint from $(\kappa_2)^{T_c \to T}|\bar{x}$.*

Proof. Denote by $\mathcal{P}^{\bar{x}}_{T_c}$ the set of polynomials $\{P^{T_c \to T} | P \in \mathtt{proj}(\kappa, \bar{x}), \kappa \in \mathcal{K}_{T_c}\}$. For each cell $\kappa_1 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E}_1)$, there exists $\tilde{\kappa}_1 \in \mathcal{K}(\mathcal{P}^{\bar{x}}_{T_c}, \mathcal{E}_1)$ such that $\kappa_1 \subseteq \tilde{\kappa}_1$. For each cell $\kappa_2 \in \mathcal{K}(\mathcal{P}'_{T_c}, \mathcal{E}_2)$, as $\mathcal{E}_1|\bar{x} = (\mathcal{E}_2)^{T_c \to T}|\bar{x}$, $(\kappa_2)^{T_c \to T}|\bar{x}$ is a union of cells in $\mathcal{K}(\mathcal{P}^{\bar{x}}_{T_c}, \mathcal{E}_1)$. So either $\tilde{\kappa}_1$ is disjoint with or contained in $(\kappa_2)^{T_c \to T}|\bar{x}$. If $\tilde{\kappa}_1$ and $(\kappa_2)^{T_c \to T}|\bar{x}$ are disjoint, then $(\kappa_2)^{T_c \to T}|\bar{x}$ and $\kappa_1|\bar{x}$ are disjoint. If $\tilde{\kappa}_1 \subseteq (\kappa_2)^{T_c \to T}|\bar{x}$, then we have $\kappa_1 \sqsubseteq \tilde{\kappa}_1 \subseteq (\kappa_2)^{T_c \to T}|\bar{x}$ so $\kappa_1 \sqsubseteq_{\bar{x}} (\kappa_2)^{T_c \to T}$. $\square$

3.6.5 Extended Isomorphism Types

Given the Hierarchical Cell Decomposition $\{\mathcal{K}_T\}_{T \in \mathcal{H}}$, we can extend our notion of isomorphism type to support arithmetic.

Definition 81. *For navigation set $\mathcal{E}_T$, equality type $\sim_\tau$ over $\mathcal{E}^+_T$ and $\kappa \in \mathcal{K}_T$, the triple $\tau = (\mathcal{E}_T, \sim_\tau, \kappa)$ is an extended T-isomorphism type if*

- *$(\mathcal{E}_T, \sim_\tau)$ is a T-isomorphism type, and*
- *$\kappa = \kappa(\sigma, \mathcal{P}'_T|(\mathcal{E}^T_{\mathbb{R}} \cap \mathcal{E}^+_T))$ for some sign condition σ of $\mathcal{P}'_T|(\mathcal{E}^T_{\mathbb{R}} \cap \mathcal{E}^+_T)$ such that for every numeric expression $e, e' \in \mathcal{E}^+_T$, $e \sim_\tau e'$ iff $\sigma(e - e') = 0$ and $e \sim_\tau 0$ iff $\sigma(e) = 0$.*

For each condition π over $\bar{x}^T$ and extended T-isomorphism type τ, $\tau \models \pi$ is defined as follows. For each polynomial inequality "$P \circ 0$" in π where $\circ \in \{<, >, =\}$, $P \circ 0$ is true iff $\sigma(P) \circ 0$ where σ is the sign condition of κ. The rest of the semantics is the same as in normal T-isomorphism type.

The projection of an extended T-isomorphism type τ on $\bar{x}^T_{\mathsf{in}}$ and $\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$ is defined in the obvious way. For $\tau = (\mathcal{E}_T, \sim_\tau, \kappa)$, we define that $\tau|\bar{x} = (\mathcal{E}_T|\bar{x}, \sim_\tau |\bar{x}, \kappa|\bar{x})$ for $\bar{x} = \bar{x}^T_{\mathsf{in}}$ or $\bar{x} = \bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$. The projection of τ on $\bar{x}^T_{\mathsf{in}}$ and $\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$ up to length k is defined analogously. The projection of every extended T-isomorphism type on $\bar{x}^T_{\mathsf{in}} \cup \bar{s}^T$ is an extended TS-isomorphism type.

To extend the definitions of local symbolic run and symbolic tree of runs, we first replace T-isomorphism type with extended T-isomorphism type and TS-isomorphism type with extended TS-isomorphism type in the original definitions. The semantics is extended with the following rules.

For two symbolic instances I and I' where the cell of I is κ and the cell of I' is κ', I' is a valid successor of I by applying service σ' if the following conditions hold in addition to the original requirements:

- if σ' is an internal service, then $\kappa|\bar{x}^T_{\mathsf{in}} = \kappa'|\bar{x}^T_{\mathsf{in}}$.
- if σ' is an opening service of $T_c \in \mathit{child}(T)$ or closing service of T, then $\kappa = \kappa'$.
- if σ' is a closing service of $T_c \in \mathit{child}(T)$, then $\kappa' \sqsubseteq \kappa$.

The counters $\bar{c}$ are updated as in transitions between symbolic instances without arithmetic. Each dimension of $\bar{c}$ corresponds to an extended TS-isomorphism type.

For each local symbolic run $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$, the following are additionally satisfied:

- $\kappa_{in} = \kappa_0|\bar{x}^T_{\mathsf{in}}$, where κ_{in} is the cell of τ_{in} and κ_0 is the cell of τ_0;
- if $\tau_{out} \neq \bot$, then $\kappa_{out} \sqsubseteq_{\bar{x}^T_{\mathsf{in}} \cup \bar{x}^T_{\mathsf{out}}} \kappa_{\gamma-1}$, where κ_{out} is the cell of τ_{out} and $\kappa_{\gamma-1}$ is the cell of $\tau_{\gamma-1}$.

In a symbolic tree of runs ***Sym***, for every two local symbolic runs $\tilde{\rho}_T = (\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$ and $\tilde{\rho}_{T_c} = (\tau'_{in}, \tau'_{out}, \{(I'_i, \sigma'_i)\}_{0 \le i < \gamma'})$ where $T_c \in \mathit{child}(T)$, if $\tilde{\rho}_{T_c}$ is connected to $\tilde{\rho}_T$ by an edge labeled with index i, then the following conditions must be satisfied in addition to the original requirements:

- for the cell κ_i of symbolic instance I_i and the cell κ_{in} of τ'_{in}, $\kappa_i \sqsubseteq \kappa_{in}^{T_c \to T}$.

- if $\tilde{\rho}_{T_c}$ is a returning local symbolic run, then for the cells κ_{out} of τ'_{out} and κ_j of I_j where j is the smallest index such that $\sigma_j = \sigma^c_{T_c}$ and $j > i$, we have that $\kappa_j \sqsubseteq_{\bar{x}_{\texttt{null}}} \kappa_{out}^{T_c \to T}$, where $\bar{x}_{\texttt{null}} = \{x | x \in \bar{x}^T_{T_c^{\uparrow}}, x \sim_{\tau_{j-1}} \texttt{null}\}$.

3.6.6 Connecting Actual Runs with Symbolic Runs

We next show that the connection between actual runs and symbolic runs established in Theorem 39 still holds for the extended local and symbolic runs. The structure of the proof is the same, so we only state the necessary modifications needed to handle arithmetic.

From Trees of Local Runs to Symbolic Trees of Runs

Given a tree of local runs ***Tree***, the construction of a corresponding symbolic tree of runs ***Sym*** can be done as follows. We first construct ***Sym*** from ***Tree*** without the cells following the construction described in the proof of the only-if part of Theorem 39. Then for each task T and symbolic instance I with extended isomorphism type τ in some local symbolic run of T, let $\mathcal{E}$ be the set of numeric expressions in τ and $v : \mathcal{E} \mapsto \mathbb{R}$ the valuation of $\mathcal{E}$ at I. Then the cell κ of I is chosen to be the unique cell in $\mathcal{K}(\mathcal{P}'_T, \mathcal{E})$ that contains v. For cells κ and κ' of two consecutive symbolic instances I and I' where the service that leads to I' is σ',

- if σ' is an internal service, by Lemma 78, as $\kappa | \bar{x}^T_{\mathsf{in}}$ and $\kappa' | \bar{x}^T_{\mathsf{in}}$ overlaps, we have $\kappa | \bar{x}^T_{\mathsf{in}} = \kappa' | \bar{x}^T_{\mathsf{in}}$,
- if σ' is an opening service, $\kappa = \kappa'$ is obvious, and
- if σ' is a closing service, let $\mathcal{E}$ be the numeric expressions of κ and $\mathcal{E}'$ be the numeric expressions of κ'. We have $\mathcal{E} \subseteq \mathcal{E}'$ so $\mathcal{P}'_T | \mathcal{E} \subseteq \mathcal{P}'_T | \mathcal{E}'$. So κ' can be written as $\kappa_1 \cap \kappa_2$ where $\kappa_1 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E})$ and $\kappa_2 \in \mathcal{K}(\mathcal{P}'_T, \mathcal{E}' - \mathcal{E})$. As the values of the preserved numeric expressions are equal in the two consecutive instances, we have $\kappa_1 = c$ so $\kappa \sqsubseteq \kappa'$.

Thus, each local symbolic run in ***Sym*** is valid. Following a similar analysis, one can verify that for every two connected local symbolic runs $\tilde{\rho}_T$ and $\tilde{\rho}_{T_c}$, the conditions for symbolic tree of runs stated in Section 3.6.5 are satisfied due to Lemma 80.

From Symbolic Trees of Runs to Trees of Local Runs

Given a symbolic tree of runs ***Sym***, we construct the tree of local runs ***Tree*** as follows. Recall that in the original proof, for each local symbolic run $\tilde{\rho}_T$, we construct the global isomorphism type Λ of $\tilde{\rho}_T$ and use Λ to construct the local run ρ_T and database instance D_T. With arithmetic, the construction of Λ remains unchanged but we use a different construction for ρ_T and D_T.

To construct ρ_T and D_T, we first define a sequence of mappings $\{p_i\}_{0\le i<\gamma}$ from the sequence of cells $\{\kappa_i\}_{0\le i<\gamma}$ of $\tilde{\rho}_T$ where each p_i is a mapping from $\mathcal{E}_T^+ \cap \mathcal{E}_{\mathbb{R}}^T$ to $\mathbb{R}$ and $\mathcal{E}_T^+$ is the extended navigation set of τ_i. Note that each p_i can be also viewed as a point in κ_i. The sequence of mappings $\{p_i\}_{0\le i<\gamma}$ determines the values of numeric expressions, as we shall see next. For each mapping p whose domain is the set of numeric expressions $\mathcal{E}$, we denote by $p|\bar{x}$ the projection of p to $\mathcal{E} \cap (\bar{x} \cup \{x_R.w | x \in \bar{x}\})$. Then $\{p_i\}_{0\le i<\gamma}$ is constructed as follows:

- First, we pick an arbitrary point (mapping) p_{in} from κ_{in} where κ_{in} is the cell of the input isomorphism type of $\tilde{\rho}_T$.

- Then, for each equivalence class $\mathcal{L}$ of life cycles in $\tilde{\rho}_T$, let $\kappa_{\mathcal{L}}$ be the cell of the last symbolic instances in the last dynamic segments of life cycles in $\mathcal{L}$. Pick a mapping $p_{\mathcal{L}} \in \kappa_{\mathcal{L}}$ such that $p_{\mathcal{L}}|\bar{x}_{\mathsf{in}}^T = p_{in}$. Such a mapping always exists because, by Lemma 78, for each $0 \le i < \gamma$, $\kappa_i|\bar{x}_{\mathsf{in}}^T = \kappa_{in}$.

- Next, for each equivalence class $\mathcal{S}$ of segments in $\mathcal{L}$, let $\kappa_{\mathcal{S}}$ be the cell of the last symbolic instance in segments in $\mathcal{S}$. Pick a mapping $p_{\mathcal{S}}$ from $\kappa_{\mathcal{S}}$ such that $p_{\mathcal{S}}|(\bar{x}_{\mathsf{in}}^T \cup \bar{s}^T) = p_{\mathcal{L}}|(\bar{x}_{\mathsf{in}}^T \cup \bar{s}^T)$. Such a mapping always exists because for each life cycle $L \in \mathcal{L}$ and I_i in L, $\kappa_{\mathcal{L}}|(\bar{x}_{\mathsf{in}}^T \cup \bar{s}^T) \sqsubseteq \kappa_i|(\bar{x}_{\mathsf{in}}^T \cup \bar{s}^T)$.

- Finally, for each segment $S = \{(I_i, \sigma_i)\}_{a\le i\le b} \in \mathcal{S}$, let $p_b = p_{\mathcal{S}}$, and for $a \le i < b$, let $p_i = p_{i+1}|\bar{x}$ where $\bar{x} = \{x | x \not\sim_{\tau_i} \texttt{null}\}$ are the preserved variables from I_i to I_{i+1}. Such mappings always exist because for each $a \le i < b$, $\kappa_{i+1} \sqsubseteq \kappa_i$.

For the sequence of mappings $\{p_i\}_{0\leq i<\gamma}$ constructed above, the following is easily shown:

Lemma 82. *For all local expressions (i, e) and (i', e') in the global isomorphism type Λ, where e and e' are numeric, $(i, e) \sim (i', e')$ implies that $p_i(e) = p_{i'}(e')$.*

Given the above property, we can construct ρ_T and D_T as follows. We first construct ρ_T and D_T as in the case without arithmetic. Then for each equivalence class $[(i, e)]$, we replace the value $[(i, e)]$ in ρ_T and D_T with the value $p_i(e)$. It is clear that Lemmas 44 and 53 still hold since the global equality type in Λ remains unchanged.

To construct the full tree of local runs ***Tree*** from the symbolic tree of runs, we perform the above construction in a top-down manner. For each local symbolic run $\tilde{\rho}_T$, we first construct $\{p_i\}_{0\leq i<\gamma}$ for the root $\tilde{\rho}_{T_1}$ of ***Sym*** using the above construction. Then recursively for each $\tilde{\rho}_T \in$ ***Sym*** and child $\tilde{\rho}_{T_c}$ connected to $\tilde{\rho}_T$ by an edge labeled with index i, we pick a mapping p_{in} from κ_{in} of $\tilde{\rho}_{T_c}$ such that $p_{in}^{T_c\to T} = p_i|\bar{x}^T_{T_c^\downarrow}$. And if $\tilde{\rho}_{T_c}$ is a returning run, we pick p_{out} from κ_{out} of $\tilde{\rho}_{T_c}$ such that $p_{out}^{T_c\to T}|\bar{x}_{\texttt{null}} = p_j|\bar{x}_{\texttt{null}}$ where j is index of the corresponding closing service $\sigma^c_{T_c}$ at $\tilde{\rho}_T$, and $\bar{x}_{\texttt{null}}$ is defined as above.

We next construct $\{p_i\}_{0\leq i<\gamma}$ of $\tilde{\rho}_{T_c}$ similarly to above, except that (1) p_{in} is given, and (2) if $\tilde{\rho}_{T_c}$ is a returning run, then for the equivalence class $\mathcal{L}$ of life cycles where $I_{\gamma-1}$ is contained in some life cycle $L \in \mathcal{L}$, we pick $p_{\mathcal{L}}$ such that $p_{\mathcal{L}}|\bar{x}^{T_c}_{\mathsf{in}} \cup \bar{x}^{T_c}_{\mathsf{out}} = p_{out}$. Then ρ_{T_c} and D_{T_c} are constructed following the above approach. The tree of local runs ***Tree*** is constructed as described in the proof of Theorem 39. Following the same approach, we can show:

Theorem 83. *For every HAS Γ and HLTL-FO property φ_f with arithmetic, there exists a symbolic tree of runs **Sym** accepted by $\mathcal{B}_\varphi$ iff there exists a tree of local runs **Tree** and database D such that **Tree** is accepted by $\mathcal{B}_\varphi$ on D.*

3.6.7 Complexity of Verification with Arithmetic

Similarly to the analysis in Appendix 3.10, it is sufficient to upper-bound the number of T-and TS-isomorphism types. To do so, we need to bound the size of $\{\mathcal{K}_T\}_{T\in\mathcal{H}}$. By the construction of each $\mathcal{K}_T$ and by Theorem 72, it is sufficient to bound the size of each $\mathcal{P}'_T$.

We denote by l the number of numeric expressions, s the number of polynomials in Γ and φ_f, d the maximum degree of these polynomials, t the maximum bitsize of the coefficients, and h the height of the task hierarchy $\mathcal{H}$. For each task T, we denote by $s(T)$ the number of polynomials in $\mathcal{P}'_T$ and $d(T)$ the maximum degree of polynomials in $\mathcal{P}'_T$.

If T is a leaf task, then $|\mathcal{P}_T| \leq s + l^2$. The number of polynomials in $\mathcal{P}^s_T$ is no more than the product of (1) the number of subsets of $\mathcal{E}^T_{\mathbb{R}}$, (2) the maximum number of non-empty cells over $(\mathcal{E}, \mathcal{P}_T|\mathcal{E})$ and (3) the maximum number of polynomials in each $\texttt{proj}(\kappa, \bar{x}^T_{\mathsf{in}} \cup \bar{s}^T)$. By Theorem 69, the number of polynomials is no more than the running time, which is bounded by $((s+l^2)\cdot d)^{O(l^2)}$. Then by Theorem 72, the number of non-empty cells over $(\mathcal{E}, \mathcal{P}_T|\mathcal{E})$ is at most $((s+l^2)\cdot d)^{O(l)}$. Thus, $|\mathcal{P}^s_T| \leq ((s+l^2)\cdot d)^{O(l^2)}$. By the same analysis, we obtain that for $\mathcal{P}'_T$, $s(T) = |\mathcal{P}'_T| \leq ((s+l^2)\cdot d)^{O(l^4)}$. Similarly, $d(T)$ can be upper-bounded by $((s+l^2)\cdot d)^{O(l^4)}$.

Next, if T is a non-leaf task, we denote by s' the size of $\mathcal{P}_T$ and by d' the maximum degree of polynomials in $\mathcal{P}_T$. We have that $s' \leq (s+l^2) + \sum_{T_c \in child(T)} 2^l (s(T_c)\cdot d(T_c))^{O(l^2)} \cdot (s(T_c)\cdot d(T_c))^{O(l)} \leq (s+l^2) + (s(T_c)\cdot d(T_c))^{O(l^2)}$, and $d' \leq \max_{T_c \in child(T)} (s(T_c)\cdot d(T_c))^{O(l^2)}$.

Following the same analysis as above, we have that both $s(T)$ and $d(T)$ are at most $((s'+l^2)\cdot d')^{O(l^4)}$. By solving the recursion, we obtain that $s(T), d(T) \leq ((s+l^2)\cdot d)^{(c\cdot l^6)^h}$ for some constant c. Then by Theorem 72, $|\mathcal{K}_T|$ is at most $(s(T)\cdot d(T))^{O(k)}$. So we have

Lemma 84. *For each task T, the number of cells in $\mathcal{K}_T$ is at most $((s+l^2)\cdot d)^{(c\cdot l^6)^h}$ for some constant c.*

The space used by the verification algorithm with arithmetic is no more than the space needed to pre-compute $\{\mathcal{K}_T\}_{T\in\mathcal{H}}$ plus the space for the VASS (repeated) reachability for each task T. By Theorem 73, for each task T, the set $\mathcal{K}_T$ can be computed in space $O\left(((s+l^2)\cdot d)^{(c\cdot l^6)^h}\right)$.

For VASS (repeated) reachability, according to the analysis in Appendix 3.10, state (repeated) reachability can be computed in $O(h^2\cdot N^2 \log^2 M \cdot 2^{c\cdot D \log D})$ space ($O(h^2\cdot N^2\log^2 M)$ w/o. artifact relation), where h is the height of $\mathcal{H}$, N is the size of (Γ, φ_f), M is the number of extended T-isomorphism types and D is the number of extended TS-isomorphism types. With arithmetic, M and D are the products of number of normal T-and TS-isomorphism types multiplied by $|\mathcal{K}_T|$ respectively. As l is less than the number of expressions whose upper bounds are obtained in Appendix 3.10, by applying Lemma 84, we obtain upper bounds for M and D for the different types of schema.

By substituting the bounds for M and D, we have the following results. Note that for Γ without artifact relations, the complexity is dominated by the space for pre-computing $\{\mathcal{K}_T\}_{T \in \mathcal{H}}$.

Theorem 85. *Let Γ be a HAS with* **acyclic** *schema and φ_f an HLTL-FO property over Γ, where arithmetic is allowed in Γ and φ_f. $\Gamma \models \varphi_f$ can be verified in $2\text{-}\exp(N^{O(h+r)})$ deterministic space. If Γ does not contain artifact relation, then $\Gamma \models \varphi_f$ can be verified in $\exp(N^{O(h+r)})$ deterministic space.*

Theorem 86. *Let Γ be a HAS with* **linearly-cyclic** *schema and φ_f an HLTL-FO property over Γ, where arithmetic is allowed in Γ and φ_f. $\Gamma \models \varphi_f$ can be verified in $O(2\text{-}\exp(N^{c_1 \cdot h^2}))$ deterministic space, where $c_1 = O(r)$. If Γ does not contain artifact relation, then $\Gamma \models \varphi_f$ can be verified in $O(\exp(N^{c_2 \cdot h^2}))$ deterministic space, where $c_2 = O(r)$.*

Theorem 87. *Let Γ be a HAS with* **cyclic** *schema and φ_f an HLTL-FO property over Γ, where arithmetic is allowed in Γ and φ_f. $\Gamma \models \varphi_f$ can be verified in $(h+2)\text{-}\exp(O(N))$ deterministic space. If Γ does not contain artifact relation, then $\Gamma \models \varphi_f$ can be verified in $(h+1)\text{-}\exp(O(N))$ deterministic space.*

3.7 Conclusion

This chapter shows decidability of verification for a rich artifact model capturing core elements of IBM's successful GSM system: task hierarchy, concurrency, database keys and foreign keys, arithmetic constraints, and richer artifact data. The extended framework requires the use of novel techniques including nested Vector Addition Systems and a variant of quantifier elimination tailored to our context. We improve significantly on previous work on verification of artifact systems with arithmetic [36], which only exhibits non-elementary upper bounds regardless of the schema shape, even absent artifact relations. In contrast, for acyclic and linearly-cyclic schemas, even in the presence of arithmetic and artifact relations, our new upper bounds are elementary (doubly-exponential in the input size and triply-exponential in the depth of the hierarchy). Moreover, the complexity of our verification algorithm gracefully reduces to PSPACE (for acyclic schema) and EXPSPACE in the hierarchy depth (for linearly-cyclic schema) when arithmetic and artifact relations are not present. The sole remaining case of nonelementary complexity occurs for arbitrary cyclic schemas. Altogether, our results provide substantial new insight and techniques for the automatic verification of artifact systems. We used the techniques developed in this chapter to implement a verifier for HAS with acyclic database schemas, that exhibits very good performance on a realistic benchmark obtained from existing sets of business process specifications and properties by extending them with data-aware features, as described in Chapter 5. This points to HAS with acyclic schemas as a sweet spot for verification, and is a strong indication of the practical potential of the approach.

3.8 Appendix: Framework and HLTL-FO

3.8.1 Proof of Theorem 22

We show that it is undecidable whether a HAS $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ satisfies an LTL(Σ) formula. The proof is by reduction from the repeated state reachability problem of VASS with reset arcs and bounded lossiness (RB-VASS) [96]. An RB-VASS extends the VASS reviewed in Section 3.5 as follows. In addition to increments and decrements of the counters, an action of RB-VASS also allows resetting the values of some counters to 0. And after each transition, the value of each counter can decrease non-deterministically by an integer value bounded by some constant c. The results in [96] (Definition 2 and Theorem 18) indicate that the repeated state reachability problem for RB-VASS is undecidable for every fixed $c \geq 0$, since the structural termination problem for Reset Petri-net with bounded lossiness can be reduced to the repeated state reachability problem for RB-VASS's. In our proof, we use RB-VASS's with $c = 1$.

Formally, a RB-VASS $\mathcal{V}$ (with lossiness bound 1 and dimension $d > 0$) is a pair (Q, A) where Q is a finite set of states and A is a set of actions of the form $(p, \bar{a}, q)$ where $\bar{a} \in \{-1, +1, r\}^d$, and $p, q \in Q$. A run of $\mathcal{V} = (Q, A)$ is a sequence $(q_0, \bar{z}_0), \ldots (q_n, \bar{z}_n)$ where $\bar{z}_0 = \bar{0}$ and for each $i \geq 0$, $q_i \in Q$, $\bar{z}_i \in \mathbb{N}^d$, and for some $\bar{a}$ such that $(q_i, \bar{a}, q_{i+1}) \in A$, and for $1 \leq j \leq d$:

- if $\bar{a}(j) \in \{-1, +1\}$, then $\bar{z}_{i+1}(j) = \bar{z}_i(j) + \bar{a}(j)$ or $\bar{z}_{i+1}(j) = \bar{z}_i(j) + \bar{a}(j) - 1$, and
- if $\bar{a}(j) = r$, then $\bar{z}_{i+1}(j) = 0$.

For a given RB-VASS $\mathcal{V} = (Q, A)$ and a pair of states $q_0, q_f \in Q$, we say that q_f is repeatedly reachable from q_0 if there exists a run $(q_0, \bar{z}_0) \ldots (q_n, \bar{z}_n) \ldots (q_m, \bar{z}_m)$ of $\mathcal{V}$ such that $q_n = q_m = q_f$ and $\bar{z}_n \leq \bar{z}_m$. As discussed above, checking whether q_f is repeatedly reachable from q_0 is undecidable.

We now show that for a given RB-VASS $\mathcal{V} = (Q, A)$ and (q_0, q_f), one can construct a HAS $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ and LTL(Σ) property Φ such that q_f is repeatedly reachable from q_0 iff $\Gamma \models \Phi$. At a high level, the construction of Γ uses d tasks to simulate the d-dimensional vector of counters. Each task is equipped with an artifact relation, and the number of elements in the artifact relation is the current value of the corresponding counter. Increment and decrement the counters are simulated by internal services of these tasks, and reset of the counters are simulated by closing and reopening the task (recall that this resets the artifact relation to empty). Then we specify in the LTL(Σ) formula Φ that the updates of the counters of the same action are grouped in sequence. Note that this requires coordinating the actions of sibling tasks, which is not possible in HLTL-FO. The construction is detailed next.

The database schema of Γ consists of a single unary relation $R(\underline{id})$. The artifact system has a root task T_1 and subtasks $\{P_0, P_1, \ldots, P_d, C_1, \ldots, C_d\}$ which form the following tasks hierarchy:

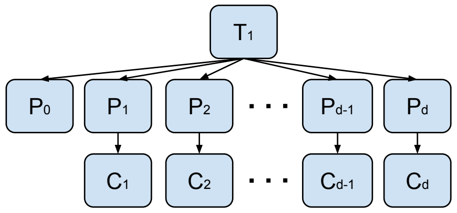


Figure 3.15. Tasks hierarchy.

The tasks are defined as follows. The root task T_1 has no variables nor internal services. The task P_0 contains a numeric variable s, indicating the current state of the RB-VASS. For each $q \in Q$, P_0 has a service σ^q, whose pre-condition is true and post-condition sets s to q.

For $i \geq 1$, task P_i has no variable. It has a single internal service σ_i^r whose pre- and post-conditions are both `true`.

Each C_i has an ID variable x, an artifact relation S_i and a pair of services σ_i^+ and σ_i^-, which simply insert x into S_i and removes an element from S_i, respectively. Intuitively, the size of S_i is the current value of the i-th counter. Application of service σ_i^r corresponds to resetting the i-th counter. And application of services σ_i^+ and σ_i^- correspond to increment and decrement of the i-th counter, respectively.

Except for the closing condition of T_1, all opening and closing conditions of tasks are `true`.

We encode the set of actions A into an LTL(Σ) formula as follows. For each state $p \in Q$, we denote by $\alpha(p)$ the set of actions starting from p. For each action $\alpha = (p, \bar{a}, q) \in A$, we construct an LTL($\Sigma$) formula $\varphi(\alpha)$ as follows. First, let $\phi_1, \dots \phi_d, \phi_{d+1}$ be LTL(Σ) formulas where:

- $\phi_{d+1} = \mathbf{X}\sigma^q$,
- for $i = d, d-1, \dots, 1$:
  - if $\bar{a}(i) = +1$, then $\phi_i = \sigma_i^+ \wedge \mathbf{X}\phi_{i+1}$,
  - if $\bar{a}(i) = -1$, then $\phi_i = (\sigma_i^- \wedge \mathbf{X}\phi_{i+1}) \vee (\sigma_i^- \wedge \mathbf{X}(\sigma_i^- \wedge \mathbf{X}\phi_{i+1}))$, and

– if $\bar{a}(i) = r$, then $\phi_i = \sigma_i^c \wedge \mathbf{X}(\sigma_i^r \wedge \mathbf{X}(\sigma_i^o \wedge \mathbf{X}\phi_{i+1}))$ where σ_i^o and σ_i^c are the opening and closing services of task C_i.

Let $\varphi(\alpha) = \mathbf{X}\phi_1$. Intuitively, $\varphi(\alpha)$ specifies a sequence of service calls that update the content of the artifact relations $S_1, \dots S_d$ according to the vector $\bar{a}$. In particular, for $\bar{a}(i) = r$, the subsequence of services $\sigma_i^c\sigma_i^r\sigma_i^o$ first closes task C_i then reopens it. This empties S_i. For $\bar{a}(i) = +1$, by executing σ_i^+, the size of S_i might be increased by 1 or 0, depending on whether the element to be inserted is already in S_i. And for $\bar{a}(i) = -1$, we let σ_i^- to be executed either once or twice, so the size of S_i can decrease by 1 or 2 nondeterministically. Then we let

$$\Phi = \Phi_{\mathtt{init}} \wedge \bigwedge_{p \in Q} \mathbf{G}\left(\sigma^p \rightarrow \bigvee_{\alpha \in \alpha(p)} \varphi(\alpha)\right) \wedge \mathbf{GF}\sigma^{q_f}$$

where $\Phi_{\mathtt{init}}$ is a formula specifying that the run is correctly initialized, which simply means that the opening services σ_T^o of all tasks are executed once at the beginning of the run, and then a σ^{q_0} is executed.

The second clause says that for every state $p \in Q$, whenever the run enters a state p (by calling σ^p), a sequence of services as specified in $\varphi(\alpha)$ is called to update $S_1, \dots, S_k$, simulating the action α that starts from p.

Finally, the last clause $\mathbf{GF}\sigma^{q_f}$ guarantees that the service σ^{q_f} is applied infinitely often, which means that q_f is reached infinitely often in the run.

We can prove the following lemma, which implies Theorem 22:

Lemma 88. *For RB-VASS (Q, A) and states $q_0, q_f \in Q$, there exists a run $(q_0, \bar{z}_0), \dots, (q_m, \bar{z}_m), \dots, (q_n, \bar{z}_n)$ of (Q, A) where $q_m = q_n = q_f$ and $\bar{z}_m \leq \bar{z}_n$ iff there exists a global run ρ of Γ such that $\rho \models \Phi$.*

3.8.2 Simplifications

We first show that the global variables, as well as set atoms, can be eliminated from HLTL-FO formulas.

Lemma 89. *Let Γ be a HAS and $\forall \bar{y} \varphi_f(\bar{y})$ an HLTL-FO formula over Γ. One can construct in linear time a HAS $\bar{\Gamma}$ and an HLTL-FO formula $\bar{\varphi}_f$, where $\bar{\varphi}_f$ contains no atoms $S^T(\bar{z})$, such that $\Gamma \models \forall \bar{y} \varphi_f(\bar{y})$ iff $\bar{\Gamma} \models \bar{\varphi}_f$.*

Proof. Consider first the elimination of global variables. Suppose Γ has tasks $T_1, \ldots, T_k$. The Hierarchical artifact system $\bar{\Gamma}$ is constructed from Γ by adding $\bar{y}$ to the variables of T_1 and augmenting the input variables of all other tasks with $\bar{y}$ (appropriately renamed). Note that $\bar{y}$ is unconstrained, so it can be initialized to an arbitrary valuation and then passed as input to all other tasks. Let $\bar{\Gamma}$ consist of the resulting tasks, $\bar{T}_1, \ldots, \bar{T}_k$. It is clear that $\Gamma \models \forall \bar{y} \varphi_f(\bar{y})$ iff $\bar{\Gamma} \models \bar{\varphi}_f$.

Consider now how to eliminate atoms of the form $S^T(\bar{z})$ from $\bar{\varphi}_f$. Recall that for all such atoms, $\bar{z} \subseteq \bar{y}$, so $\bar{z}$ is fixed throughout each run. The idea is keep track of the membership of $\bar{z}$ in S^T using two additional numeric artifact variables $x_{\bar{z}}$ and $y_{\bar{z}}$, such that $x_{\bar{z}} = y_{\bar{z}}$ indicates that $S^T(\bar{z})$ holds[7]. Specifically, a pre-condition ensures that $x_{\bar{z}} \neq y_{\bar{z}}$ initially holds, then $x_{\bar{z}} \neq y_{\bar{z}}$ is enforced as soon as there is an insertion $+S^T(\bar{s}^T)$ for which $\bar{s}^T = \bar{z}$, and $x_{\bar{z}} \neq y_{\bar{z}}$ is enforced again whenever there is a retrieval of a tuple equal to $\bar{z}$. This can be achieved using pre-and-post conditions of services carrying out the insertion or retrieval. Then the atom $S^T(\bar{z})$ can be replaced in $\bar{\varphi}_f$ with $(x_{\bar{z}} = y_{\bar{z}})$. $\square$

We next consider two simplifications of artifact systems regarding the interaction of tasks with their subtasks.

Lemma 90. *Let Γ be a HAS and φ_f an HLTL-FO property over Γ. One can construct a HAS $\tilde{\Gamma}$ and an HLTL-FO formula $\tilde{\varphi}_f$ such that $\Gamma \models \varphi_f$ iff $\tilde{\Gamma} \models \tilde{\varphi}_f$ and: (i) $\bigcup_{T_c \in child(T)} \bar{x}^T_{T_c^\uparrow}$*

[7]This is done to avoid introducing constants, that could also be used as flags.

and $\bigcup_{T_c \in child(T)} \bar{x}^T_{T_c^{\downarrow}}$ *are disjoint for each task* T *in* $\tilde{\Gamma}$, (ii) *for each child task* $T_c \in child(T)$, $\bar{x}^T_{T_c^{\uparrow}} \cap \text{VAR}_{\text{val}} = \emptyset$.

Proof. Consider (i). We describe here informally the construction of $\tilde{\Gamma}$ that eliminates overlapping between $\bigcup_{T_c \in child(T)} \bar{x}^T_{T_c^{\uparrow}}$ and $\bigcup_{T_c \in child(T)} \bar{x}^T_{T_c^{\downarrow}}$. For each task T and for each subtask T_c of T, for each variable $x \in \bar{x}^T_{T_c^{\downarrow}}$, we introduce to T a new variable $\hat{x}$ whose type is the same as the type (id or numeric) of x. We denote by $\hat{x}^T_{T_c\downarrow}$ the set of variables added to T for subtask T_c. Then instead of passing $\bar{x}^T_{T_c^{\downarrow}}$ to T_c, T passes $\hat{x}^T_{T_c\downarrow}$ to T_c when T_c opens. And for the opening service $\sigma^o_{T_c}$ with opening condition π, we check π in conjunction with $\bigwedge_{x \in \bar{x}^T_{T_c^{\downarrow}}} (x = \hat{x})$. Note that $\bigcup_{T_c \in child(T)} \hat{x}^T_{T_c\downarrow}$ and $\bigcup_{T_c \in child(T)} \bar{x}^T_{T_c^{\uparrow}}$ are disjoint. By this construction, in each run of $\tilde{\Gamma}$, after each application of an internal service σ of task T, the variables in $\hat{x}^T_{T_c\downarrow}$ for each subtask T_c receives a set of non-deterministically chosen values. Then each subtask T_c can be opened only when $\hat{x}^T_{T_c\downarrow}$ and $\bar{x}^T_{T_c^{\downarrow}}$ have the same values. So passing $\hat{x}^T_{T_c\downarrow}$ to T_c is equivalent to passing $\bar{x}^T_{T_c^{\downarrow}}$ to T_c.

To guarantee that there is a bijection from the runs of Γ to the runs of $\tilde{\Gamma}$, we also need to make sure that the values of $\hat{x}^T_{T_c\downarrow}$ are non-deterministically chosen before the first application of internal service. (Recall that they either contain 0 or `null` at the point when T is opened.) So we extend T with an extra binary variable x_{init} and an extra internal service σ^{init}_T. Variable x_{init} indicates whether task T has been "initialized". The service σ^{init}_T has precondition that checks whether $x_{\text{init}} = 0$ and post-condition sets $x_{\text{init}} = 1$. It sets all id variables to `null` and numeric variables 0 except for variables in $\hat{x}^T_{T_c\downarrow}$ for any T_c. So application of σ^{init}_T assigns values to $\hat{x}^T_{T_c\downarrow}$ for every subtask T_c non-deterministically and all other variables are initialized to the initial state when T is opened. All other services are modified such that they can be applied only when $x_{\text{init}} = 1$ and initialize $\hat{x}^T_{T_c\downarrow}$ with non-deterministically chosen values for all subtask T_c. So in a projected run ρ_T of $\tilde{\Gamma}$, the suffix with $x_{\text{init}} = 1$ corresponds to the original projected run of Γ. Thus we only need to rewrite the HLTL-FO property φ_f to $\tilde{\varphi}_f$ such that each formula in Φ_T only looks at the suffix of projected run ρ_T after x_{init} is set to be 1 (namely, each $\psi \in \Phi_T$ is replaced with $\mathbf{F}((x_{\mathsf{init}} = 1) \wedge \psi))$.

Now consider (ii). We outline the construction of $\tilde{\Gamma}$ and $\tilde{\varphi}_f$ informally. For each task T, we introduce a set of new numeric variables $\{x_{T_c} | T_c \in child(T), x \in \bar{x}^T_{T_c^\uparrow} \cap \mathit{VAR}_{val}\}$ to $\bar{x}^T$. Intuitively, these variables contain non-deterministically guessed returning values from each child task T_c. These are passed to each child task T_c as additional input variables. Before T_c returns, these are compared to the values of the returning numeric variables of T_c, and T_c returns only if they are identical. More formally, for each child task T_c of T, variables $\{x_{T_c} | x \in \bar{x}^T_{T_c^\uparrow} \cap \mathit{VAR}_{val}\}$ are passed from T to T_c as part of the input variables of T_c. For each variable x_{T_c} in T, we let $x_{T_c \to T} \in \bar{x}^{T_c}$ be the corresponding input variable of x_{T_c}. And for each x_{T_c}, we denote by x_{ret} the variable in $\bar{x}^{T_c}$ satisfying that $f_{out}(x) = x_{ret}$ for f_{out} in the original Γ. Then at T_c, we remove all numeric variables from $\bar{x}^{T_c}_{ret}$ and add condition $\bigwedge_{x \in \bar{x}^T_{T_c^\uparrow} \cap \mathit{VAR}_{val}} x_{ret} = x_{T_c \to T}$ to the closing condition of T_c. Note that we need to guarantee that the variables in $\{x_{T_c} | T_c \in child(T), x \in \bar{x}^T_{T_c^\uparrow} \cap \mathit{VAR}_{val}\}$ obtain non-deterministically guessed values. This can be done as in the simulation for (i).

Conditions on $\bar{x}^T$ after a subset T's children has returned are evaluated using the guessed values for the variables returned so far. Specifically, the correct value to be used is the latest returned by a child transaction, if any (recall that children tasks can overwrite each other's numeric return variables in the parent). Keeping track of the sequence of returned transactions and evaluating conditions with the correct value can be easily done directly in the verification algorithm, at negligible extra cost. This means that we can assume that tasks have the form in (ii) without the exponential blowup in the conditions, but with a quadratic blowup in the number of variables.

To achieve the simulation fully via the specification is costlier because some of the conditions needed have exponential size. We next show how this can be done. Intuitively, we guess initially an order of the return of the children transactions and enforce that it be respected. We also keep track of the children that have already returned. Let *child*$(T) = \{T_1, \ldots, T_n\}$. To guess an order of return, we use new ID variables $\bar{o} = \{o_{ij} \mid 1 \leq i, j \leq n\}$. Intuitively,

$o_{ij} \neq \texttt{null}$ says that T_i returns before T_j. We also use new ID variables $\{t_i \mid 1 \leq i \leq n\}$, where $t_i \neq \texttt{null}$ means that T_i has returned. The variables $\bar{o}$ are subject to a condition specifying the axioms for a total order:

$$\begin{aligned}
&\wedge_{1 \leq i,j \leq n}(o_{ij} \neq \texttt{null} \vee o_{ji} \neq \texttt{null}) \\
&\wedge_{1 \leq i < j \leq n} \neg(o_{ij} \neq \texttt{null} \wedge o_{ji} \neq \texttt{null}) \\
&\wedge_{1 \leq i,j,m \leq n}((o_{ij} \neq \texttt{null} \wedge o_{jm} \neq \texttt{null}) \rightarrow o_{im} \neq \texttt{null})
\end{aligned}$$

These are enforced using pre-conditions of services as well as one additional initial internal service (which in turn requires a minor modification to φ_f, similarly to (i)). When T_i returns, t_i is set to a non-null value, and the condition

$$\bigwedge_{1 \leq i,j \leq n} (t_i \neq \texttt{null} \wedge t_j = \texttt{null}) \rightarrow o_{ij} \neq \texttt{null}$$

enforcing that transactions return in the order specified by $\bar{o}$ is maintained using pre-conditions. Observe that, at any given time, the latest transaction that has returned is the T_i such that

$$t_i \neq \texttt{null} \wedge \bigwedge_{1 \leq j \leq n} ((o_{ij} \neq \texttt{null}) \rightarrow t_j = \texttt{null})$$

For each formula π over $\bar{x}^T$, we construct a formula $o(\pi)$ by replacing each variable $x \in \bar{x}^T_{\mathbb{R}}$ with x_{T_c} for the latest T_c where $x \in \bar{x}^T_{T_c^{\uparrow}}$ if there is such T_c). The size of the resulting $o(\pi)$ is exponential in the maximum arity of database relations. Finally we obtain $\tilde{\Gamma}$ and $\tilde{\varphi}_f$ by replacing, for every $T \in \mathcal{H}$, each condition π over $\bar{x}^T$ with $o(\pi)$. One can easily verify that $\tilde{\Gamma} \models \tilde{\varphi}_f$ iff $\Gamma \models \varphi_f$ and for every task T of Γ, $\bar{x}^T_{T_c^{\uparrow}}$ does not contain numeric variables. This completes the proof of (ii). $\square$

The construction in (i) takes linear time in the original specification and property. For (ii), the construction introduces a quadratic number of new variables and the size of conditions becomes exponential in the maximum arity of data-base relations. However, as discussed in Appendix 3.8, the verification algorithm can be slightly adapted to circumvent the blowup in the specification without penalty to the complexity. Intuitively, this makes efficient use of non-determinism, avoiding the explicit enumeration of choices required in the specification, which leads to the exponential blowup.

3.8.3 Proof of Theorem 29

For conciseness, we refer throughout the proof to *propositionally* interleaving-invariant LTL-FO simply as interleaving-invariant LTL-FO.

Showing that HLTL-FO expresses only interleaving-invariant LTL-FO properties is straightforward. The converse however is non-trivial. We begin by showing a normal form for LTL formulas, which facilitates the application to our context of results from [54, 55] on temporal logics for concurrent processes. Consider the alphabet $\mathbf{H}(\Gamma) = \{(\kappa, \sigma) \mid (\kappa, stg, \sigma) \in \mathbf{A}(\Gamma)\}$. Thus, $\mathbf{H}(\Gamma)$ is $\mathbf{A}(\Gamma)$ with the stage information omitted. Let $\mathcal{H}(\Gamma) = h(\mathcal{L}(\Gamma))$ where $h((\kappa, stg, \sigma)) = (\kappa, \sigma)$. We define local-LTL to be LTL using the set of propositions $\mathrm{P}\Sigma = \{(p, \sigma) \mid p \in P_T, \sigma \in \Sigma_T^{obs}\}$. A proposition (p, σ) holds in $(\bar{\kappa}, \bar{\sigma})$ iff $\bar{\sigma} = \sigma$ and $\bar{\kappa}(p)$ is true. The definition of interleaving-invariant local-LTL formula is the same as for LTL.

Lemma 91. *For each interleaving-invariant LTL formula φ over $\mathcal{L}(\Gamma)$ one can construct an interleaving-invariant local-LTL formula $\bar{\varphi}$ over $\mathcal{H}(\Gamma)$ such that for every $u \in \mathcal{L}(\Gamma)$, $u \models \varphi$ iff $h(u) \models \bar{\varphi}$ where $h((\kappa, stg, \sigma)) = (\kappa, \sigma)$.*

Proof. We use the equivalence of FO and LTL over ω-words [82]. It is easy to see that each LTL formula φ over $\mathcal{L}(\Gamma)$ can be translated into an FO formula $\psi(\varphi)$ over $\mathcal{H}(\Gamma)$ using only propositions in $\mathrm{P}\Sigma$, such that for every $u \in \mathcal{L}(\Gamma)$, $u \models \varphi$ iff $h(u) \models \psi(\varphi)$. Indeed, it is straightforward to define by FO means the stage of each transaction in a given configuration, as well as each proposition in $P \cup \Sigma$ in terms of propositions in $\mathrm{P}\Sigma$, on words in $\mathcal{H}(\Gamma)$. One can then construct from the FO sentence $\psi(\varphi)$ an LTL formula $\bar{\varphi}$ equivalent to it over words in

$\mathcal{H}(\Gamma)$, using the same set of propositions PΣ. The resulting LTL formula is thus in local-LTL, and it is easily seen that it is interleaving-invariant. $\square$

We use the propositional variant HLTL of HLTL-FO, whose semantics over ω-words in $\mathcal{H}(\Gamma)$ is defined similarly to the semantics of HLTL-FO on global runs. Recall that in HLTL, the LTL formulas applying to transaction T use propositions in $P_T \cup \Sigma_T^{obs}$ and expressions $[\psi]_{T_c}$ where T_c is a child of T and ψ is an HLTL formula applying to T_c.

We show the following key fact.

Lemma 92. *For each interleaving-invariant local-LTL formula over $\mathcal{H}(\Gamma)$ there exists an equivalent HLTL formula over $\mathcal{H}(\Gamma)$.*

Proof. To show completeness of HLTL, we use a logic shown in [54, 55] to be complete for expressing LTL properties invariant with respect to valid interleavings of actions of concurrent processes (or equivalently, well-defined on Mazurkievicz traces). The logic, adapted to our framework, operates on partial orders $\preceq_u$ of words $u \in \mathcal{H}(\Gamma)$, and is denoted LTL($\preceq$). For $u = \{(\kappa_i, \sigma_i) \mid i \geq 0\}$, we define the projection of u on T as the subsequence $\pi_T(u) = \{(\kappa_{i_j}|_{P_T}, \sigma_{i_j})\}_{j \geq 0}$ where $\{\sigma_{i_j} \mid j \geq 0\}$ is the subsequence of $\{\sigma_i \mid i \geq 0\}$ retaining all services in Σ_T^{obs}. LTL($\preceq$) uses the set of propositions PΣ and the following temporal operators on $\preceq_u$:

- $\mathbf{X}_T\varphi$, which holds in (κ_i, σ_i) if $\pi_T(v) \neq \epsilon$ for $v = \{(\kappa_j, \sigma_j) \mid j \geq m\}$, where m is a minimum index such that $i \prec_u m$, and φ holds on $\pi_T(v)$;

- $\varphi \mathbf{U}_T \psi$, which holds in (κ_i, σ_i) if $\pi_T(v) \neq \epsilon$ for $v = \{(\kappa_j, \sigma_j) \mid j \geq i\}$, and $\varphi \mathbf{U} \psi$ holds on $\pi_T(v)$.

From Theorem 18 in [54] and Proposition 2 and Corollary 26 in [55] it follows that LTL($\preceq$) expresses all local-LTL properties over $\mathcal{H}(\Gamma)$ invariant with respect to interleavings.

We next show that HLTL can simulate LTL($\preceq$). To this end, we consider an extension of HLTL in which LTL($\preceq$) formulas may be used in addition to propositions in $P_T \cup \Sigma_T^{obs}$ in every formula applying to transaction T. We denote the extension by HLTL+LTL($\preceq$). Note that each formula ξ in LTL($\preceq$) is an HLTL+LTL($\preceq$) formula. The proof consists in showing that the LTL($\preceq$) formulas can be eliminated from HLTL+LTL($\preceq$) formulas. This is done by recursively reducing the depth of nesting of X_T and U_T operators, and finally eliminating propositions. We define the *rank* of an LTL($\preceq$) formula to be the maximum number of X_T and U_T operators along a path in its syntax tree. For a formula ξ in HLTL+LTL($\preceq$), we define $r(\xi) = (n, m)$ where n is the maximum rank of an LTL($\preceq$) formula occurring in ξ, and m is the number of such formulas with rank n. The pairs (n, m) are ordered lexicographically.

Let ξ be an HLTL+LTL($\preceq$) formula. For uniformity of notation, we define $[\xi]_{T_1} = \xi$. We associate to ξ the tree *Tree*(ξ) with root $[\xi]_{T_1}$, whose nodes are all occurrences of subformulas of the form $[\psi]_T$ in ξ, with an edge from $[\psi_i]_{T_i}$ to $[\psi_j]_{T_j}$ if the latter occurs in ψ_i and T_j is a child of T_i in $\mathcal{H}$.

Consider an HLTL+LTL($\preceq$) formula ξ such that $r(\xi) \geq (1, 1)$. Suppose ξ has a subformula $\mathbf{X}_T\varphi$ in LTL($\preceq$) of maximum rank. Pick one such occurrence and let $\bar{T}$ be the minimum task (wrt $\mathcal{H}$) such that $\mathbf{X}_T\varphi$ occurs in $[\psi]_{\bar{T}}$. We construct an HLTL+LTL($\preceq$) formula $\bar{\xi}$ such that $r(\bar{\xi}) < r(\xi)$, essentially by eliminating $\mathbf{X}_T$. We consider 4 cases: $T = \bar{T}$, T is a descendant or ancestor of $\bar{T}$, or neither.

Suppose first that $T = \bar{T}$. Consider an occurrence of $\mathbf{X}_T\varphi$. Intuitively, there are two cases: $\mathbf{X}_T\varphi$ is evaluated inside the run of T corresponding to $[\psi]_T$, or at the last configuration. In the first case ($\neg\sigma_T^c$ holds), $\mathbf{X}_T\varphi$ is equivalent to $\mathbf{X}\varphi$. In the second case (σ_T^c holds), $\mathbf{X}_T\varphi$ holds iff φ holds at the next call to T. Thus, ξ is equivalent to $\xi_1 \vee \xi_2$, where:

1. ξ_1 says that φ does not hold at the next call to T (or no such call exists) and $\mathbf{X}_T\varphi$ is replaced in ψ by $\neg\sigma_T^c \wedge \mathbf{X}\varphi$

2. ξ_2 says that φ holds at the next call to T (which exists) and $\mathbf{X}_T\varphi$ is replaced in ψ by $\neg\sigma_T^c \rightarrow \mathbf{X}\varphi$.

We next describe how ξ_1 states that φ does not hold at the next call to T (ξ_2 is similar). We need to state that either there is no future call to T, or such a call exists and $\neg\varphi$ holds at the first such call. Consider the path from T_1 to T in $\mathcal{H}$. Assume for simplicity that the path is $T_1, T_2, \ldots, T_k$ where $T_k = T$. For each i, $1 \leq i < k$, we define inductively (from $k-1$ to 1) formulas $\alpha_i, \beta_i(\neg\varphi)$ such that α_i says that there is no call leading to T in the remainder of the current subrun of T_i, and $\beta_i(\neg\varphi)$ says that such a call exists and the first call leads to a subrun of T satisfying $\neg\varphi$. First, $\alpha_{k-1} = \mathbf{G}(\neg\sigma^o_{T_k})$ and $\beta_{k-1}(\neg\varphi) = \neg\sigma^o_{T_k}\ \mathbf{U}\ [\neg\varphi]_{T_k}$. For $1 \leq i < k-1$, $\alpha_i = \mathbf{G}(\sigma^o_{T_{i+1}} \rightarrow [\alpha_{i+1}]_{T_{i+1}})$ and $\beta_i(\neg\varphi) = (\sigma^0_{T_{i+1}} \rightarrow [\alpha_{i+1}]_{T_{i+1}})\ \mathbf{U}\ [\beta_{i+1}(\neg\varphi)]_{T_{i+1}}$. Now $\xi_1 = \xi_1^0 \vee \bigvee_{1 \leq j < k} \xi_1^j$ where ξ_1^0 states that there is no next call to T and ξ_1^j states that T_j is the minimum task such that the next call to T occurs during the *same* run of T_j (and satisfies $\neg\varphi$). More precisely, let $[\psi_1]_{T_1}, [\psi_2]_{T_2}, \ldots [\psi_k]_{T_k}$ be the path leading from $[\xi]_{T_1}$ to $[\psi]_T$ in *Tree*(ξ) (so $\psi_1 = \xi$ and $\psi_k = \psi$). Then ξ_1^0 is obtained by replacing each ψ_i by $\bar{\psi}_i$, $1 \leq i < k$, defined inductively as follows. First, $\bar{\psi}_{k-1}$ is obtained from ψ_{k-1} by replacing $[\psi_k]_{T_k}$ with $[\psi_k]_{T_k} \wedge \alpha_{k-1}$. For $1 \leq i < k-1$, $\bar{\psi}_i$ is obtained from ψ_i by replacing $[\psi_{i+1}]_{T_{i+1}}$ with $[\bar{\psi}_{i+1}]_{T_{i+1}} \wedge \alpha_i$. For $1 \leq j < k$, ξ_1^j is obtained by replacing in ψ_j, $[\psi_{j+1}]_{T_{j+1}}$ with $[\bar{\psi}_{j+1}]_{T_{j+1}} \wedge \beta_j(\neg\varphi)$. It is clear that ξ_1 states the desired property. The formula ξ_2 is constructed similarly. Note that $r(\xi_1 \vee \xi_2) < r(\xi)$.

Now suppose T is an ancestor of $\bar{T}$. We reduce this case to the previous ($T = \bar{T}$). Let T' be the child of T. Suppose $[\psi_T]_T$ is the ancestor of $[\psi]_{\bar{T}}$ in *Tree*(ξ). Then ξ is equivalent to $\bar{\xi} = \xi_1 \vee \xi_2$ where:

1. ξ_1 says that φ does not hold at the next action of T wrt $\preceq$ (or no such next action exists) and ψ is replaced by $\psi(\mathbf{X}_T\varphi \leftarrow \textit{false})$ ($\leftarrow$ denotes substitution)

2. ξ_2 says that φ holds at the next action of T wrt $\preceq$ and ψ is replaced by $\psi(\mathbf{X}_T\varphi \leftarrow \textit{true})$

To state that φ does not hold at the next call to T (or no such call exists) ξ_1 is further modified by replacing in ψ_T, $[\psi_{T'}]_{T'}$ with $[\psi_{T'}]_{T'} \wedge (\mathbf{G}(\neg\sigma^c_{T'}) \vee (\neg\sigma^c_{T'}\ \mathbf{U}\ (\sigma^c_{T'} \wedge \neg\mathbf{X}_T\varphi))$. Smilarly, ξ_2 is further modified by replacing in ψ_T, $[\psi_{T'}]_{T'}$ with $[\psi_{T'}]_{T'} \wedge (\neg\sigma^c_{T'}\ \mathbf{U}\ (\sigma^c_{T'} \wedge \mathbf{X}_T\varphi))$. Note that there are now two occurrences of $\mathbf{X}_T\varphi$ in the modified ψ_T's. By applying twice the construction for the case $\bar{T} = T$ we obtain an equivalent $\bar{\xi}$ such that $r(\bar{\xi}) < r(\xi)$.

Next consider the case when $\bar{T}$ is an ancestor of T. Suppose the path from T_1 to T in $\mathcal{H}$ is $T_1, \ldots, T_i, \ldots T_k$ where $T_i = \bar{T}$ and $T_k = T$. Consider the value of $\mathbf{X}_T\varphi$ in the run ρ_ψ of $\bar{T}$ on which ψ is evaluated. Similarly to the case $T = \bar{T}$, there are two cases: φ holds at the next invocation of T following ρ_ψ, or it does not. Thus, ξ is equivalent to $\xi_1 \vee \xi_2$, where:

1. ξ_1 says that φ does not hold at the next call to T (or no such call exists) and $\mathbf{X}_T\varphi$ is replaced in ψ by $\beta_i(\varphi)$, where $\beta_i(\varphi)$ says that there exists a future call leading to T in the current run of $\bar{T}$, and the first such run of T satisfies φ; $\beta_i(\varphi)$ is constructed as in the case $T = \bar{T}$.

2. ξ_2 says that φ holds at the next call to T following the current run of $\bar{T}$ and $\mathbf{X}_T\varphi$ is replaced in ψ by $\alpha_i \vee \beta_i(\varphi)$ where α_i, constructed as for the case $T = \bar{T}$, says that there is no future call leading to T in the current run of $\bar{T}$.

To say that φ does not hold at the next call to T following ρ_ψ (or no such call exists), ξ_1 is modified analogously to the case $\bar{T} = T$, and similarly for ξ_2.

Finally suppose the least common ancestor of $\bar{T}$ and T is $\hat{T}$ distinct from both. Let $[\psi_{\hat{T}}]_{\hat{T}}$ be the ancestor of $[\psi]_{\bar{T}}$ in *Tree*(ξ). Consider the value of $\mathbf{X}_T\varphi$ in the run of $\bar{T}$ on which ψ is evaluated. There are two cases: φ holds at the next invocation of T following the run of $\bar{T}$, or it does not. Thus, ξ is equivalent to $\xi_1 \vee \xi_2$, where:

1. ξ_1 says that φ does not hold at the next call to T (or no such call exists) and ψ is replaced by $\psi(\mathbf{X}_T\varphi \leftarrow \textit{false})$

2. ξ_2 says that φ holds at the next call to T and ψ is replaced by $\psi(\mathbf{X}_T\varphi \leftarrow \textit{true})$

To say that φ does not hold at the next call to T (or no such call exists), ξ_1 is modified analogously to the case $\bar{T} = T$, and similarly for ξ_2, taking into account the fact that the next call to T, if it exists, must take place in the current run of $\hat{T}$ or of one of its ancestors. This completes the simulation of $\mathbf{X}_T\varphi$.

Now suppose ξ has a subformula $(\varphi_1 \mathbf{U}_T \varphi_2)$ of maximum rank. Pick one such occurrence and let $\bar{T}$ be the minimum task (wrt $\mathcal{H}$) such that $(\varphi_1 \mathbf{U}_T \varphi_2)$ occurs in $[\psi]_{\bar{T}}$. There are several cases: $\bar{T} = T$, $\bar{T}$ is an ancestor or descendant of T, or neither. The simulation technique is similar to the above. We outline the construction for the most interesting case when $\bar{T} = T$.

Consider the run of T on which $[\psi]_T$ is evaluated. There are two cases: $(\dagger)$ $(\varphi_1 \mathbf{U}_T \varphi_2)$ holds on the concatenation of the future runs of T, or $(\dagger)$ does not hold. Thus, ξ is equivalent to $\xi_1 \vee \xi_2$ where:

1. ξ_1 says that $(\dagger)$ holds and ψ is modified by replacing the occurrence of $(\varphi_1 \mathbf{U}_T \varphi_2)$ with $\mathbf{G}\varphi_1 \vee (\varphi_1 \mathbf{U} \varphi_2)$, and

2. ξ_2 says that $(\dagger)$ does not hold and ψ is modified by replacing the occurrence of $(\varphi_1 \mathbf{U}_T \varphi_2)$ with $(\varphi_1 \mathbf{U} \varphi_2)$.

We show how ξ_1 ensures $(\dagger)$. Let $T_1, \ldots, T_k$ be the path from root to T in $\mathcal{H}$. For each i, $1 \leq i < k$, we define inductively (from $k-1$ to 1) formulas α_i, β_i as follows. Intuitively, α_i says that all future calls leading to T from the current run of T_i must result in runs satisfying $\mathbf{G}\ \varphi_1$:

- $\alpha_{k-1} = \mathbf{G}(\sigma^o_{T_k} \rightarrow [\mathbf{G}\ \varphi_1]_{T_k})$,
- for $1 \leq i < k-1$, $\alpha_i = \mathbf{G}(\sigma^o_{T_{i+1}} \rightarrow [\alpha_{i+1}]_{T_{i+1}})$

The formula β_i says that there must be a future call to T in the current run of T_i satisfying $\varphi_1 \mathbf{U} \varphi_2$ and all prior calls result in runs satisfying $\mathbf{G}\varphi_1$:

- $\beta_{k-1} = (\sigma^o_{T_k} \rightarrow [\mathbf{G}\varphi_1]_{T_k})\ \mathbf{U}\ [\varphi_1 \mathbf{U} \varphi_2]_{T_k}$,
- for $1 \leq i < k-1$, $\beta_i = (\sigma^o_{T_{i+1}} \rightarrow [\alpha_{i+1}]_{T_{i+1}})\ \mathbf{U}\ [\beta_{i+1}]_{T_{i+1}}$.

Now ξ_1 is $\bigvee_{1 \leq j < k} \xi_j$ where ξ_j states that the concatenation of runs resulting from calls to T within the run of T_j on which $[\psi_j]_{T_j}$ is evaluated, satisfies $(\varphi_1 \mathbf{U} \varphi_2)$. More precisely, let $[\psi_1]_{T_1}, \ldots, [\psi_k]_{T_k}$ be the path from $[\xi]_{T_1}$ to $[\psi]_T$ in *Tree*(ξ) (so $\psi_1 = \xi$ and $\psi_k = \psi$). For each j we define ψ^j_i, $1 \leq i < k$ as follows:

- if $j < k-1$, ψ^j_{k-1} is obtained from ψ_{k-1} by replacing $[\psi_k]_{T_k}$ with $[\psi_k]_{T_k} \wedge \alpha_{k-1}$

- if $j = k-1$, ψ^j_{k-1} is obtained from ψ_{k-1} by replacing $[\psi_k]_{T_k}$ with $[\psi_k]_{T_k} \wedge \beta_{k-1}$
- for $j < i < k-1$, ψ^j_i is obtained from ψ_i by replacing $[\psi^j_{i+1}]_{T_{i+1}}$ with $[\psi^j_{i+1}]_{T_{i+1}} \wedge \alpha_i$
- ψ^j_j is obtained from ψ_j by replacing $[\psi^j_{j+1}]_{T_{j+1}}$ with $[\psi^j_{j+1}]_{T_{j+1}} \wedge \beta_j$
- for $1 \leq i < j$, ψ^j_i is obtained from ψ_i by replacing $[\psi_{i+1}]_{T_{i+1}}$ with $[\psi^j_{i+1}]_{T_{i+1}}$.

Finally, $\xi_j = [\psi^j_1]_{T_1}$. The formula ξ_2 is constructed along similar lines. This completes the case $(\varphi_1 \mathbf{U}_T \varphi_2)$.

Consider now the case when the formula of maximum rank is a proposition $(p, \sigma) \in \mathrm{P}\Sigma$, where $p \in P_T$ and $\sigma \in \Sigma^{obs}_T$. There are several cases:

- (p, σ) occurs in $[\psi]_T$. Then (p, σ) is replaced with $p \wedge \sigma$.
- (p, σ) occurs in $[\psi]_{\bar{T}}$ where $\bar{T} \neq T$ and $\bar{T}$ is not a child or parent of T. Then (p, σ) is replaced with *false*.
- (p, σ) occurs in $[\psi_{T'}]_{T'}$ for some parent T' of T. If $\sigma \in \Sigma_T$ then (p, σ) is replaced with *false* in $\psi_{T'}$. If $\sigma = \sigma^o_T$ then (p, σ) is replaced by $[p]_T$. If $\sigma = \sigma^c_T$, we use the past temporal operator **S** whose semantics is symmetric to **U**. This can be simulated in LTL, again as a consequence of Kamp's Theorem [82]. The proposition (p, σ) is replaced in $\psi_{T'}$ by $\sigma^c_T \wedge ((\neg\sigma^o_T)\ \mathbf{S}\ [\mathbf{F}(\sigma^c_T \wedge p)]_T)$
- (p, σ) occurs in $[\psi_{T'}]_{T'}$ for some child T' of T. Let $[\psi_T]_T$ be the parent of $[\psi_{T'}]_{T'}$ in *Tree*(ξ). As above, if $\sigma \in \Sigma_T$ then (p, σ) is replaced with *false* in $\psi_{T'}$. If $\sigma = \sigma^o_{T'}$, there are two cases: (1) p holds in T when the call to T' generating the run on which $\psi_{T'}$ is evaluated is made, and (2) the above is false. Thus, ψ_T is replaced by $\psi^1_T \vee \psi^2_T$ where ψ^1_T corresponds to (1) and ψ^2_T to (2). Specifically:
  - ψ^1_T is obtained from ψ_T by replacing $[\psi_{T'}]_{T'}$ with $p \wedge [\psi^1_{T'}]_{T'}$, where $\psi^1_{T'}$ is obtained from $\psi_{T'}$ by replacing $(p, \sigma^o_{T'})$ with $\sigma^o_{T'}$

– ψ_T^2 is obtained from ψ_T by replacing $[\psi_{T'}]_{T'}$ with $\neg p \wedge [\psi_{T'}^2]_{T'}$ where $\psi_{T'}^2$ is obtained from $\psi_{T'}$ by replacing $(p, \sigma_{T'}^o)$ with *false*.

Now suppose $\sigma = \sigma_{T'}^c$. Again, there are two cases: (1) if T' returns then p holds in the run of T when T' returns, and (2) this is false. The two cases are treated similarly to the above.

This concludes the proof of the lemma. $\square$

Theorem 29 now follows. Let φ_f be an interleaving-invariant LTL-FO formula over Γ. By Lemma 91, we can assume that φ is in local-LTL and in particular uses the set of propositions PΣ. By Lemma 92, there exists an HLTL formula ξ equivalent to φ over ω-words in $\mathcal{H}(\Gamma)$, using propositions in $P \cup \Sigma$. Moreover, by construction, each sub-formula $[\psi]_T$ of ξ uses only propositions in $P_T \cup \Sigma_T^{obs}$. It is easily seen that ξ_f is a well-formed HLTL-FO formula equivalent to φ_f on all runs of Γ.

3.9 Appendix: Restrictions and Undecidability

We provide a proof of Theorem 30 for relaxing restriction (2). Recall that HAS$^{(2)}$ allows subtasks of a given task to overwrite non-null ID variables. The same proof idea can be used for restrictions (1) to (7).

Proof. We show undecidability by reduction from the Post Correspondence Problem (PCP) [106, 115]. Given an instance $P = \{(a_i, b_i)\}_{1 \leq i \leq k}$ of PCP, where each (a_i, b_i) is a pair of non-empty strings over $\{0, 1\}$, we show how to construct a HAS$^{(2)}$ Γ and HLTL-FO formula φ_f such that there is a solution to P iff there exists a run of Γ satisfying φ_f (i.e., $\Gamma \not\models \neg\varphi_f$).

The database schema of Γ contains a single relation

$$G(\underline{id}, \texttt{next}, \texttt{label})$$

where `next` is a foreign-key attribute referencing attribute id and `label` is a non-key attribute. Let α, β be distinct id values in G. A *path* in G from α to β is a sequence of IDs $i_0, \ldots, i_n$

in G where $\alpha = i_0$, $\beta = i_n$, and for each $j, 0 \leq j < n$, $i_{j+1} = i_j.\texttt{next}$. It is easy to see that there is at most one path from α to β for which $i_j \neq \alpha, \beta$ for $0 < j < n$, and the path must be simple ($i_0, i_1, \ldots, i_n$ are distinct). If such a path exists, we denote by $w(\alpha, \beta)$ the sequence of labels $i_0.\texttt{label}, \ldots, i_n.\texttt{label}$ (a word over $\{0, 1\}$, assuming the values of `label` are 0 or 1). Intuitively, Γ and φ_f do the following given database G:

1. non-deterministically pick two distinct ids α, β in G
2. check that there exists a simple path from α to β and that $w(\alpha, \beta)$ witnesses a solution to P; the uniqueness of the simple path from α to β is essential to ensure that $w(\alpha, \beta)$ is well defined.

Step 2 requires simultaneously parsing $w(\alpha, \beta)$ as $a_{s_1} \ldots a_{s_m}$ and $b_{s_1} \ldots b_{s_m}$ for some $s_i \in [1, k], 1 \leq i \leq m$, by synchronously walking the path from α to β with two pointers P_a and P_b. More precisely, P_a and P_b are initialized to α. Then repeatedly, an index $s_j \in [1, k]$ is picked non-deterministically, and P_a advances $|a_{s_j}|$ steps to a new position P'_a, such that the sequence of labels along the path from P_a to P'_a is a_{s_j} and no id along the path equals α or β. Similarly, P_b advances $|b_{s_j}|$ steps to a new position P'_b, such that the sequence of labels along the path from P_b to P'_b is b_{s_j} and no id along the path equals α or β. This step repeats until P_a and P_b simultaneously reach β (if ever). The property φ_f checks that eventually $P_a = P_b = \beta$, so $w(\alpha, \beta)$ witnesses a solution to P.

In more detail, we use two tasks T_p and T_c where T_c is a child task of T_p (see Figure 3.16).

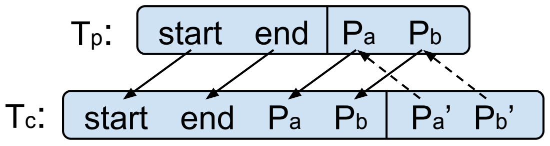


Figure 3.16. Undecidability for HAS$^{(2)}$.

Task T_p has two input variables *start*, *end* (initialized to distinct ids α and β by the global precondition), and two artifact variables P_a and P_b (holding the two pointers). T_p also has a binary artifact relation S whose set variables are (P_a, P_b). At each segment of T_p, the subtask T_c is called with $(P_a, P_b, \textit{start}, \textit{end})$ passed as input. Then an internal service of T_c computes P'_a and P'_b, such that P_a, P'_a, P_b and P'_b satisfy the condition stated above for some $s_j \in [1, k]$. Then T_c closes and returns P'_a and P'_b to T_p, overwriting P_a and P_b (note that this is only possible because restriction (2) is lifted). At this point we would like to call T_c again, but multiple calls to a subtasks are disallowed between internal transitions. To circumvent this, we equip T_p with an internal service that simply propagates $(P_a, P_b, \textit{start}, \textit{end})$. The variables $\textit{start}, \textit{end}$ are automatically propagated as input variables of T_p. Propagating (P_a, P_b) is done by inserting it into S and retrieving it in the next configuration (so $\delta = \{+S(P_a, P_b), -S(P_a, P_b)\}$). Now we are allowed to call again T_c, as desired.

It can be shown that there exists a solution to P iff there exists a run of the above system that reaches a configuration in which $P_a = P_b = \textit{end}$. This can be detected by a second internal service *success* of T_p with pre-condition $P_a = P_b = \textit{end}$. Thus, the HLTL-FO property φ_f is simply $[\mathbf{F}\ (\textit{success})]_{T_p}$. Note that this is in fact an HLTL formula. Thus, checking HLTL-FO (and indeed HLTL) properties of HAS$^{(2)}$ systems is undecidable. □

3.10 Appendix: Complexity of Verification without Arithmetic

Let Γ be a HAS and φ_f an HLTL-FO formula over Γ. Recall the VASS $\mathcal{V}(T, \beta)$ constructed for each task T and assignment β to Φ_T. According to the discussion of the complexity of verification in Section 3.5, checking whether $\Gamma \not\models \varphi_f$ can be done in $O(h \log n \cdot 2^{c \cdot d \log(d)})$ nondeterministic space, where c is a constant, h is the depth of $\mathcal{H}$, and n, d bound the number of states, resp. vector dimensions of $\mathcal{V}(T, \beta)$ for all T and β. We will estimate these bounds using the maximum number of T-isomorphism types, denoted M, and the maximum number of TS-isomorphism types, denoted D. We also denote by N the size of (Γ, φ_f). To complete the analysis, the specific bounds M and D will be computed for acyclic, linear-cyclic, and cyclic schemas, as well as with and without artifact relations.

By our construction, the vector dimension of each $\mathcal{V}(T,\beta)$ is the number of TS-isomorphism types, so bounded by D. The number of states is at most the product of the number of distinct T-isomorphism types, the number states in $B(T,\beta)$, the number of all possible $\bar{o}$ and the number of possible states of $\bar{c}_{ib}$. And since the number of T_c-isomorphism types is no more than the number of T-isomorphism types if T_c is child of T, the number of all possible $\bar{o}$ is at most $(3+M)^{|child(T)|} \leq (3+M)^N$. Note that the number of states in $B(T,\beta)$ is at most exponential in the size of the HLTL-FO property φ_f (extending the classical construction [125]). Thus, $n = M \cdot 2^{O(N)} \cdot (3+M)^N \cdot 2^D$ bounds the number of states of all $\mathcal{V}(T,\beta)$. It follows that $O(h \log n \cdot 2^{c \cdot d \log(d)}) = O(h \cdot N \cdot \log M \cdot 2^{c \cdot D \cdot \log D})$, yielding the complexity of checking $\Gamma \not\models \varphi_f$. Thus, checking whether $\Gamma \models \varphi_f$ can be done in $O(h^2 \cdot N^2 \log^2 M \cdot 2^{c \cdot D \log D})$ deterministic space by Savitch's Theorem [115], for some constant c.

For artifact systems with no artifact relation, the bounds degrade to $O(h \cdot N \log M)$ and $O(h^2 \cdot N^2 \log^2 M)$.

The number of T- and TS-isomorphism types depends on the type of the schema $\mathcal{DB}$ of Γ, as described next. In our analysis, we denote by r the number of relations in $\mathcal{DB}$ and a the maximum arity of relations in $\mathcal{DB}$. We also let $k = \max_{T \in \mathcal{H}} |\bar{x}^T|$, $s = \max_{T \in \mathcal{H}} |\bar{s}^T|$ and h be the height of $\mathcal{H}$.

Acyclic schema. if $\mathcal{DB}$ is acyclic, then the length of each expression in the navigation set is bounded by the number of relations in $\mathcal{DB}$. So the size of the navigation set of each T-isomorphism type is at most $a^r k$. The total number of T-isomorphism types is at most the product of the number of possible navigation sets and the number of possible equality types. So $M = (r+1)^k \cdot (a^r k)^{a^r k}$ is a bound for the number of T-isomorphism types for every T.

For TS-isomorphism types, we note that within the same path in $\mathcal{V}(T,\beta)$, all TS-isomorphism types have the same projections on $\bar{x}^T_{\text{in}}$ since the input variables are unchanged throughout a local symbolic run. So within each query of (repeated) reachability, each TS-

isomorphism type can be represented by (1) the equality connections from expressions starting with $x \in \bar{x}^T_{\mathsf{in}}$ to expressions starting with $x \in \bar{s}^T$ and (2) the equality connections within expressions starting with $x \in \bar{s}^T$. For (1), the total number of all possible connections is at most ${M_1}^{M_2}$ where M_1 is the number of expressions starting with $x \in \bar{x}^T_{\mathsf{in}}$ and M_2 is the number of expressions starting with $x \in \bar{s}^T$. For (2), the total number of all possible connections is at most $M_2^{M_2}$. Note that $M_1 \leq a^r k$ and $M_2 \leq a^r s$. So the total number of TS-isomorphism type is at most $D = (r+1)^s \cdot (a^r k \cdot a^r s)^{a^r s} = (r+1)^s \cdot (a^{2r} k \cdot s)^{a^r s}$. So for $\mathcal{DB}$ of fixed size and S^T of fixed arity, the number of T-isomorphism type is exponential in k and the number of TS-isomorphism type is polynomial in k.

By substituting the above values of M and D in the space bound $O(h^2 \cdot N^2 \log^2 M \cdot 2^{c \cdot D \log D})$, we obtain:

Theorem 93. *For HAS Γ with acyclic schema and HLTL-FO property φ_f over Γ, $\Gamma \models \varphi_f$ can be checked in $O(\exp(N^{c_1}))$ deterministic space, where $c_1 = O(a^{r \log r} s)$. If Γ does not contain artifact relations, then $\Gamma \models \varphi_f$ can be checked in $c_2 \cdot N^{O(1)}$ deterministic space, where $c_2 = O(a^{2r} \log^2 a^r)$.*

Note that if $\mathcal{DB}$ is a Star schema [84, 126], which is a special case of acyclic schema, then the size of the navigation set is at most ark instead of $a^r k$. So verification has the complexities stated in Theorem 93, with constants $c_1 = O(ars)$ and $c_2 = O(ar^2 \log^2 ar)$ respectively.

Note that with the simulation used in Lemma 90, the number of variables is at most quadratic in the original number of variables. This only affects the constants in the above complexities.

Linearly-cyclic schema. Consider the case where $\mathcal{DB}$ is linearly cyclic. To bound the number of T- and TS-isomorphism types, it is sufficient to bound $h(T)$, which equals to $1 + k \cdot F(\delta)$ where $\delta = \max_{T_c \in child(T)}\{h(T_c)\}$ if T is a non-leaf task and $\delta = 1$ if T is a leaf. And recall that $F(\delta)$ is the maximum number of distinct paths of length at most δ starting from any relation in the foreign key graph FK. If $\mathcal{DB}$ is linearly cyclic, then by definition, the graph of cycles in FK form an acyclic graph G (each node in G is a cycle in the FK graph and there is an edge from cycle u to cycle v iff there is an edge from some node in u to some node in v in FK).

Consider each path P of length at most δ in FK. P can be decomposed into a list of subsequences of nodes, where each subsequence consists of nodes within the same cycle in FK (as shown in Figure 3.17).

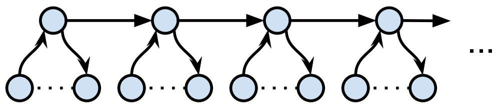


Figure 3.17. A path in a Linearly-Cyclic Foreign Key graph.

So $F(\delta)$ can be bounded by the product of (1) the number of distinct paths in G starting from any cycle and (2) the maximum number distinct paths of length at most δ formed using subsequences of nodes from cycles within the same path in G. It is easy to see that (1) is at most a^r. And since the length of a path in G is at most r, (2) is at most δ^r. Thus $F(\delta)$ is bounded by $a^r \cdot \delta^r = (a \cdot \delta)^r$.

So if $\mathcal{DB}$ is linearly cyclic, then $h(T)$ is bounded by $1 + a^r k$ if T is a leaf task and $h(T)$ is bounded by $1 + (a \cdot \delta)^r \cdot k$ if T is non-leaf task where $\delta = \max_{T_c \in child(T)}\{h(T_c)\}$. By solving the recursion, for every task T, we have that $h(T) \leq c \cdot (a \cdot k)^{r \cdot h}$ for some constant c. So the size of the navigation set of each T-isomorphism type is at most $c \cdot (a \cdot k)^{r(h+1)}$. Thus the number of T- and TS-isomorphism types are bounded by $(r+1)^k \cdot (c \cdot (a \cdot k)^{r(h+1)})^{c \cdot (a \cdot k)^{r(h+1)}}$. By an analysis similar to that for acyclic schemas, we can show that

Theorem 94. *For HAS Γ with linearly-cyclic schema and HLTL-FO property φ_f over Γ, $\Gamma \models \varphi_f$ can be checked in $O(2\text{-}\exp(N^{c_1 \cdot h}))$ deterministic space where $c_1 = O(r)$. If Γ does not contain artifact relations, then $\Gamma \models \varphi_f$ can be checked in $O(N^{c_2 \cdot h})$ deterministic space where $c_2 = O(r)$.*

Cyclic schema. If $\mathcal{DB}$ is cyclic, then each relation in FK has at most a outgoing edges so $F(\delta)$ is bounded by a^δ. So $h(T) = O(k \cdot a^\delta)$ where $\delta = 1$ if T is a leaf task and

$\delta = \max_{T_c \in child(T)} h(T_c)$ otherwise. Solving the recursion yields $h(T) = h\text{-}\exp(O(N))$. By pursuing the analysis similarly to the above, we obtain the following:

Theorem 95. *For HAS Γ with cyclic schema and HLTL-FO property φ_f over Γ, $\Gamma \models \varphi_f$ can be checked in $(h+2)\text{-}\exp(O(N))$ deterministic space. If Γ does not contain artifact relations, then $\Gamma \models \varphi_f$ can be checked in $h\text{-}\exp(O(N))$ deterministic space.*

To summarize, the schema type determines the size of the navigation set, and hence the complexity of verification, as follows (h the height of the task hierarchy and N the size of (Γ, φ_f)).

- Acyclic schemas are the least general, yet sufficiently expressive for many applications. A special case of acyclic schema is the Star schema [84, 126] (or Snowflake schema) which is widely used in modeling business process data. For fixed acyclic schemas, the navigation sets have constant depth.

- Linearly-cyclic schemas extend acyclic schemas but yield higher complexity. In general, the size of the navigation set is exponential in h and polynomial in N. Linearly-cyclic schemas allow very simple cyclic foreign key relations such as a single Employee-Manager relation. They include important special cases such as schemas where each relation has at most one foreign key attribute.

- Cyclic schemas allow arbitrary foreign keys but also come with much higher complexity (a tower of exponentials of height h), as the size of navigation sets become hyper-exponential wrt h.

Acknowledgement

This chapter contains material from “Verification of Hierarchical Artifact Systems” by Alin Deutsch, Yuliang Li and Victor Vianu, which appears in Proceedings of the 35th ACM SIGMODSIGACT- SIGAI Symposium on Principles of Database Systems (PODS 2016) [45]. The dissertation author was the primary investigator of this paper.

Chapter 4

SpinArt: A Spin-based Verifier for Artifact Systems

4.1 Overview

In this chapter, we study the implementation of SpinArt, a practical artifact system verifier based on the classic model checker Spin. We begin by formally defining Tuple Artifact System (TAS) reviewed informally in Chapter 2. Compared to the reviewed TAS model, the version of TAS considered here is slightly different, with additional restriction on the schema. The model captures a core fragment of HAS that can potentially be handled by Spin (Section 4.2). At a high level, a TAS consists of an acyclic read-only database, a tuple of updatable artifact variables and a set of services specifying transitions of the system. The properties of TAS's to be verified are specified using LTL-FO.

This model is expressive enough to allow data of unbounded domain and size, which are features not directly supported by Spin or other state-of-the-art model checkers. Therefore, a direct translation into Spin requires setting limits on the size of the data and its domain, resulting in an incomplete verifier. To address this challenge, we exploit the symbolic verification techniques establishing the decidability results for HAS and develop a simple algorithm for translating TAS specifications and properties into equivalent problem instances that can be verified by Spin, *without sacrificing either the soundness or the completeness of the verifier*. However, a naive use of Spin still results in poor performance even with the translation algorithm.

Therefore, we develop an array of nontrivial optimizations techniques to render verification tractable. To the best of our knowledge, SpinArt is the first implementation of an artifact system verifier that preserves decidability under unbounded data while being based on off-the-shelf model checking technology. The main contributions are summarized as follows.

- We formally define Tuple Artifact System (TAS), a core fragment of HAS that permits efficient implementation of a Spin-based verifier. By exploiting the symbolic verification approach from previous work [45, 36], we show a simple algorithm for translating the verification problem into an equivalent instance in Spin. This algorithm forms the basis of our implementation of SpinArt.
- We implement SpinArt with two nontrivial optimization techniques to achieve satisfactory performance. The first consists of a more efficient translation algorithm avoiding a quadratic blowup in the size of the specification due to keys and foreign keys, so that it shortens significantly the compilation and execution time for Spin. The second optimization is based on static analysis, and greatly reduces the size of the search space by exploiting constraints extracted from the input specification during a pre-computation phase. Although these techniques are designed with Spin as the target tool, we believe that they can be adapted to implementations based on other off-the-shelf model checkers.
- We evaluate the performance of SpinArt experimentally using real-world data-driven work-flows and properties. We created a benchmark of artifact systems and LTL-FO properties from existing sets of business process specifications and temporal properties by extending them with data-aware features. The experiments highlight the impact of the optimizations and various parameters of the specifications and properties on the performance of SpinArt.

This chapter is organized as follows. We start by reviewing in Section 4.2 the HAS model and formally defining TAS, a core fragment of HAS. We also define the variant of LTL-FO used in this chapter to specify properties of TAS's. In Section 4.3 we first review the theory adapted to TAS, then describe the initial direct implementation of SpinArt based on the symbolic representation technique for HAS. We next present the specialized optimizations, essential for achieving acceptable performance. The experimental results are shown in Section 4.4. Finally, we discuss and conclude in Section 4.5.

4.2 The Model

In this section, we present the variant of artifact systems supported by our verifier, as well as the temporal logic LTL-FO used to specify the properties to be verified.

4.2.1 Tuple Artifact Systems

The model is a variant of the Hierarchical Artifact System (HAS) model presented in Chapter 3 and is restricted as follows:

- it disallows evolving relations in artifact data
- it does not use arithmetic in service pre-and-post conditions
- the underlying database schema uses an *acyclic* set of foreign keys

The implemented model retains the hierarchy of tasks present in HAS. However, for simplicity of exposition, we only define formally the core of the model, consisting of a single task in which a tuple of artifact values evolves throughout the workflow under the action of services. For clarity, we also describe the algorithms in terms of the core model. The exposition can be easily extended to a hierarchy of tasks.

We now present the syntax and semantics of TAS. The formal definitions below are illustrated with an intuitive example of the TAS specification of an order fulfillment business process originally written in BPMN [1]. Intuitively, the workflow allows customers to place orders and the supplier company to process the orders.

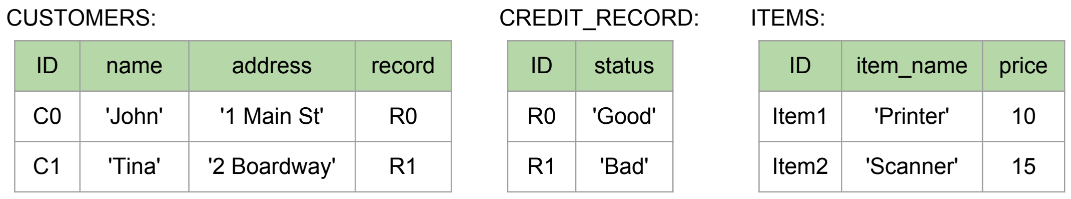
At a high level, a TAS consists of a database schema and a TAS artifact schema. The database schema for TAS is the same as the acyclic database schema for HAS (Definition 3). Here, we illustrate the acyclic schema with an example.

Example 96. *The order fulfillment workflow has the following database schema:*

- $\mathtt{CUSTOMERS}(\mathrm{ID}, \mathtt{name}, \mathtt{address}, \mathtt{record})$, $\quad \mathtt{ITEMS}(\mathrm{ID}, \mathtt{item\_name}, \mathtt{price})$
  $\mathtt{CREDIT\_RECORD}(\mathrm{ID}, \mathtt{status})$

The IDs *are key attributes,* $\mathtt{price}$*,* $\mathtt{item\_name}$*,* $\mathtt{name}$*,* $\mathtt{address}$*,* $\mathtt{status}$ *are non-key attributes, and* $\mathtt{record}$ *is a foreign key attribute satisfying the dependency* $\mathtt{CUSTOMERS}[record] \subseteq \mathtt{CREDIT\_RECORD}[\mathrm{ID}]$*. Intuitively, the* $\mathtt{CUSTOMERS}$ *table contains customer information with a foreign key pointing to the customers' credit records stored in* $\mathtt{CREDIT\_RECORD}$*. The* $\mathtt{ITEMS}$ *table contains information on the items. Note that the schema is acyclic as there is only one foreign key reference from* $\mathtt{CUSTOMERS}$ *to* $\mathtt{CREDIT\_RECORD}$*.*

Figure 5.2 shows an example of an instance of the acyclic schema of the order fulfillment workflow. Note that the domains of $\mathtt{CUSTOMERS}.\mathrm{ID}$*,* $\mathtt{ITEMS}.\mathrm{ID}$ *and* $\mathtt{CREDIT\_RECORD}.\mathrm{ID}$ *and the domain for non-key attributes are mutually disjoint. The domain of* $\mathtt{CUSTOMERS}.record$ *is included in* $Dom(\mathtt{CREDIT\_RECORD}.\mathrm{ID})$ *since* $\mathtt{record}$ *is a foreign key attribute referencing* $\mathtt{CREDIT\_RECORD}.\mathrm{ID}$*.*



CUSTOMERS:

ID	name	address	record
C0	'John'	'1 Main St'	R0
C1	'Tina'	'2 Boardway'	R1

CREDIT_RECORD:

ID	status
R0	'Good'
R1	'Bad'

ITEMS:

ID	item_name	price
Item1	'Printer'	10
Item2	'Scanner'	15

Figure 4.1. An instance of an acyclic schema.

We next proceed with the definition of TAS artifact schema. Similarly to the database schema, we consider two infinite, disjoint sets VAR_{id} of ID variables and VAR_{val} of data variables. We associate to each variable x its domain $Dom(x)$. If $x \in \mathit{VAR}_{id}$, then $Dom(x) = \mathit{DOM}_{id} \cup \{\mathtt{null}\}$, and if $x \in \mathit{VAR}_{val}$, then $Dom(x) = \mathit{DOM}_{val} \cup \{\mathtt{null}\}$. An *artifact variable* is a variable in $\mathit{VAR}_{id} \cup \mathit{VAR}_{val}$. If $\bar{x}$ is a sequence of artifact variables, a *valuation* of $\bar{x}$ is a mapping ν associating to each variable x in $\bar{x}$ an element in $Dom(x)$.

Definition 97. *A **TAS artifact schema** is a pair $\mathcal{A} = \langle \mathcal{DB}, \bar{x} \rangle$ with an acyclic database schema $\mathcal{DB}$ and $\bar{x} \subseteq \text{VAR}_{id} \cup \text{VAR}_{\text{val}}$ a set of artifact variables. The domain of each variable $x \in \bar{x}$ is either $\text{DOM}_{val} \cup \{\texttt{null}\}$ or $dom(R.\text{ID}) \cup \{\texttt{null}\}$ for some relation $R \in \mathcal{DB}$. In the latter case we say that the type of x is $\mathtt{type}(x) = R.\text{ID}$. An **instance** ρ of $\mathcal{A}$ is a pair (D, ν) where D is a finite instance of $\mathcal{DB}$ and ν is a valuation of $\bar{x}$.*

Example 98. *The TAS artifact schema of the order fulfillment example consists of the acyclic database schema described in Example 96 and the following artifact variables:*

- *ID variables:* `cust_id` *of type* `CUSTOMERS`.ID *and* `item_id` *of type* `ITEMS`.ID
- *Non-ID variables:* `status` *and* `instock`

Intuitively, `cust_id` *and* `item_id` *store the ID of the customer and the ID of the item ordered by the customer. Variable* `status` *indicates the different stages of the order, namely "Init", "OrderPlaced", "Passed" (passed the credit check), "Shipped" or "Failed". Variable* `instock` *indicates whether the ordered item is in stock.*

For a given TAS artifact schema $\mathcal{A} = \langle \mathcal{DB}, \bar{x} \rangle$ and a sequence $\bar{y}$ of variables, a *condition* on $\bar{y}$ is a quantifier-free first-order (FO) formula over $\mathcal{DB} \cup \{=\}$ whose variables are included in $\bar{y}$. In more detail, a condition over $\bar{y}$ is a Boolean combination of relational or equality atoms whose variables are included in $\bar{y}$. A relational atom over relation $R(ID, A_1, \ldots, A_m, F_1, \ldots, F_n) \in \mathcal{DB}$, is of the form $R(x, y_1, \ldots, y_m, z_1, \ldots, z_n)$, where $\{x, z_1, \ldots, z_n\} \subseteq \mathit{VAR}_{id}$ and $\{y_1, \ldots, y_m\} \subseteq \mathit{VAR}_{val}$. An equality atom is of the form $x = z$, where x is variable and z is a variable of the same type, or $x \in \mathit{VAR}_{val}$ and $z \in \mathit{DOM}_{val}$. The special constant `null` can be used in equalities. If α is a condition on $\bar{y} \subseteq \bar{x}$, D an instance of $\mathcal{DB}$ and ν a valuation of $\bar{x}$, we denote by $D \models \alpha(\nu)$ the fact that D satisfies α with valuation ν, with standard semantics. For an atom $R(\bar{z})$ in α where $R \in \mathcal{DB}$, if $\nu(z) = \texttt{null}$ for some $z \in \bar{z}$, then $R(\nu(\bar{z}))$ is false (since the database instances do not contain `null`). Although conditions are quantifier-free, conditions with existentially quantified variables (denoted $\exists$FO) can be easily simulated by adding variables to $\bar{x}$, so we use them as shorthand whenever convenient.

Example 99. *The following $\exists$FO condition states that the customer with ID* `cust_id` *has good credit:*

$$\exists n \exists a \exists r\ \texttt{CUSTOMERS}(\texttt{cust\_id}, n, a, r) \wedge \texttt{CREDIT\_RECORD}(r, \text{“\textit{Good}”}).$$

We next define services in TAS.

Definition 100. *Let $\mathcal{A} = \langle \mathcal{DB}, \bar{x} \rangle$ be a TAS artifact schema. A **TAS service** σ of $\mathcal{A}$ is a tuple $\langle \pi, \psi, \bar{y} \rangle$ where:*

- π *and* ψ*, called* pre-condition *and* post-condition*, respectively, are conditions over* $\bar{x}$*, and*
- $\bar{y}$ *is the set of* propagated variables*, where* $\bar{y} \subseteq \bar{x}$*.*

Intuitively, π and ψ are conditions which must be satisfied by the previous and the next instance respectively when σ is applied. In addition, the values stored in $\bar{y}$ are propagated to the next instance.

Example 101. *The order fulfillment TAS has the following five services: **EnterCustomer**, **EnterItem**, **CheckCredit**, **Restock** and **ShipItem**. Intuitively, for each order, the workflow first obtains the customer and item information by applying the **EnterCustomer** service and the **EnterItem** service. Then the credit record of the customer is checked by the **CheckCredit** service. If the record is good, **ShipItem** can be called to ship the item to the customer. If the requested item is unavailable, then **Restock** must be called before **ShipItem** to procure the item.*

*Next, we illustrate each service in more detail. The **EnterCustomer** and **EnterItem** allow the customer to enter his/her information and the ordered item's information. The* `CUSTOMERS` *and* `ITEMS` *tables are queried to obtain the customer ID and item ID. When **EnterItem** is called, the supplier also checks whether the item is currently in stock and sets the variable* `instock` *to "Yes" or "No" accordingly. This step is modeled as an external service so we use the post-condition to enforce that the two values are chosen nondeterministically. In both services, if both* `cust_id` *and* `item_id` *have been entered, the current status of the order is updated to "OrderPlaced" (otherwise it remains "Init"). The two services can be called multiple* *times to allow the customer to modify previously entered data. The propagated variables of* ***EnterCustomer*** *are* `item_id` *and* `instock` *since their values are not modified when the service is applied. Similarly, the only propagated variable of* ***EnterItem*** *is* `cust_id`*. The two services are formally specified in Fig. 4.2, and Fig. 5.5 shows transitions that result from applying the two services consecutively.*

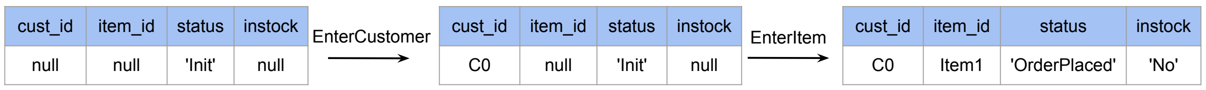
EnterCustomer:
Pre-condition: $\texttt{status} = \text{“Init”}$
Propagated: $\{\texttt{item\_id}, \texttt{instock}\}$
Post-condition:

$\exists n \exists a \exists r\ \texttt{CUSTOMERS}(\texttt{cust\_id}, n, a, r) \wedge$
$(\texttt{item\_id} \neq \texttt{null} \rightarrow$
$\texttt{status} = \text{“OrderPlaced”}) \wedge$
$(\texttt{item\_id} = \texttt{null} \rightarrow \texttt{status} = \text{“Init”})$

EnterItem:
Pre-condition: $\texttt{status} = \text{“Init”}$
Propagated: $\{\texttt{cust\_id}\}$
Post-condition:

$\exists n \exists p\ \texttt{ITEMS}(\texttt{item\_id}, n, p) \wedge$
$(\texttt{instock} = \text{“Yes”} \vee \texttt{instock} = \text{“No”}) \wedge$
$(\texttt{cust\_id} \neq \texttt{null} \rightarrow \texttt{status} = \text{“OrderPlaced”}) \wedge$
$(\texttt{cust\_id} = \texttt{null} \rightarrow \texttt{status} = \text{“Init”})$

Figure 4.2. Examples of two TAS services.



cust_id	item_id	status	instock
null	null	'Init'	null

cust_id	item_id	status	instock
C0	null	'Init'	null

cust_id	item_id	status	instock
C0	Item1	'OrderPlaced'	'No'

Figure 4.3. Two transitions caused by TAS services.

We describe in brief the rest of the TAS services. The ***CheckCredit*** *service can be called if* $\texttt{status} = $ *“OrderPlaced”. It checks the credit record of the customer using the condition IsGood(*`cust_id`*) in Example 99. If the credit record is good, then it updates* `status` *to “Passed” otherwise to “Failed”. The* ***Restock*** *service can be called if* $\texttt{status} = $ *“Passed” which means that the credit check is passed. The service simply updates* `instock` *to “Yes”, indicating that ordered item is now in stock. Finally, the* ***ShipItem*** *can be called if* $\texttt{status} = $ *“Passed” and* $\texttt{instock} = $ *“Yes”. It updates* `status` *to “Shipped”, meaning that the shipment is successful.*

We can now define TAS's.

Definition 102. *A* ***Tuple Artifact System*** *(TAS) is a triple* $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$*, where* $\mathcal{A}$ *is a TAS* *artifact schema, Σ is a set of TAS services over $\mathcal{A}$, and Π, called the global pre-condition, is a condition over $\bar{x}$.*

We next define the semantics of TAS. Intuitively, a run of a TAS on a database D consists of an infinite sequence of transitions among artifact instances (also referred to as configurations, or snapshots), starting from an initial artifact tuple satisfying pre-condition Π. We begin by defining single transitions.

Definition 103. *Let $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ be a tuple artifact system, where $\mathcal{A} = \langle \bar{x}, \mathcal{DB} \rangle$. We define the transition relation among instances of $\mathcal{A}$ as follows. For two instances $(\nu, D), (\nu', D')$ and TAS service $\sigma = \langle \pi, \psi, \bar{y} \rangle$, $(\nu, D) \xrightarrow{\sigma} (\nu', D')$ if $D = D'$, $D \models \pi(\nu)$, $D \models \psi(\nu')$, and $\nu'(y) = \nu(y)$ for each $y \in \bar{y}$.*

Then a *run* of the TAS $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ on database instance D is an infinite sequence $\rho = \{(I_i, \sigma_i)\}_{i \geq 0}$, where each I_i is an instance (ν_i, D) of $\mathcal{A}$, $D \models \Pi(\nu_0)$, and for each $i > 0$, $I_{i-1} \xrightarrow{\sigma_i} I_i$. In the run, σ_0 is a special initializing service *init*, whose role is to produce the instance I_0.

4.2.2 Specifying Properties of TAS's with LTL-FO

In this chapter we focus on verifying temporal properties of runs of a tuple artifact system. For instance, in the business process of the example above, we would like to specify properties such as:

(†) If an order is taken and the ordered item is out of stock, then the item must be restocked before it is shipped.

Such temporal properties of a TAS are specified using a variant of LTL-FO. Formally, an LTL-FO property of a tuple artifact system $\mathcal{A}$ is obtained starting from an LTL formula using some set $P \cup \Sigma$ of propositions. Propositions in P are interpreted as conditions over the variables $\bar{x}$ together with some additional *global* variables $\bar{y}$, shared by different conditions and allowing to refer to the state of the task at different moments in time. The global variables are universally quantified over the entire property. A proposition $\sigma \in \Sigma$ indicates the application of service σ in a given transition. LTL-FO formulas are defined as follows.

Definition 104. *Let $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ be a TAS where $\mathcal{A} = (\bar{x}, \mathcal{DB})$. Let $\bar{y}$ be a finite sequence of variables in* $\text{VAR}_{id} \cup \text{VAR}_{\text{val}}$ *disjoint from $\bar{x}$, called* global variables. *An LTL-FO formula for Γ is an expression $\forall \bar{y} \varphi_f$, where:*

- *φ is an LTL formula with propositions $P \cup \Sigma$, where P is a finite set of proposition disjoint from Σ*
- *f is a function from P to conditions over $\bar{x} \cup \bar{y}$*
- *φ_f is obtained by replacing each $p \in P$ with $f(p)$*

For example, suppose we wish to specify property (†). The property is of the form $\varphi = \mathbf{G}(p \rightarrow (\neg q \,\mathbf{U}\, r))$, which means: if p happens, then in the future q will not happen until r is true. Here p says that the **EnterItem** service is called and chooses an out-of-stock item, q states that the **ShipItem** service is called with the same item, and r states that the service **Restock** is called to restock the item. Since the item mentioned in p, q and r must be the same, the formula requires using a global variable i denoting the ID of the item. This yields the following LTL-FO property:

$$\forall i \;\; \mathbf{G}((\texttt{EnterItem} \wedge \texttt{item\_id} = i \wedge \texttt{instock} = \text{“No”}) \rightarrow$$
$$(\neg(\texttt{ShipItem} \wedge \texttt{item\_id} = i) \;\mathbf{U}\; (\texttt{Restock} \wedge \texttt{item\_id} = i)))$$

A correct specification can enforce (†) simply by requiring in the pre-condition of **ShipItem** that the item is in stock. One such pre-condition is ($\texttt{instock} = \text{“Yes”} \wedge \texttt{status} = \text{“Passed”}$), meaning that the item is in stock and the customer passed the credit check. However, in a similar specification where $\texttt{instock} = \text{“Yes”}$ is not tested in the pre-condition but performed in the post-condition of **ShipItem** (i.e. the post-condition requires that if $\texttt{instock} = \text{“Yes”}$, then `status` stays unchanged so the item is not shipped), the LTL-FO property (†) is violated because

ShipItem can still be called without first calling the **Restock** service. The verifier would detect this error and produce a counter-example illustrating the violation.

We say that a run $\rho = \{(I_i, \sigma_i)\}_{i \geq 0}$ satisfies $\forall \bar{y} \varphi_f$, where $prop(\varphi) = P \cup \Sigma$, if φ is satisfied, for all valuations of $\bar{y}$ in $DOM_{id} \cup DOM_{val} \cup \{\texttt{null}\}$, by the sequence of truth assignments to $P \cup \Sigma$ induced by f on the sequence $\{(I_i, \sigma_i)\}_{i \geq 0}$. More precisely, for $p \in P$, the truth value induced for p in (I_i, σ_i) is the truth value of the condition $f(p)$ in I_i; a proposition $\sigma \in \Sigma$ holds in (I_i, σ_i) if $\sigma_i = \sigma$. A TAS Γ satisfies $\forall \bar{y} \varphi_f(\bar{y})$ if for every run ρ of Γ and valuation ν of $\bar{y}$, ρ satisfies $\varphi_f(\nu(\bar{y}))$.

It is easily seen that for given Γ with artifact variables $\bar{x}$ and LTL-FO formula $\forall \bar{y} \varphi_f(\bar{y})$, one can construct Γ' with artifact variables $\bar{x} \cup \bar{y}$ such that $\Gamma \models \forall \bar{y} \varphi_f(\bar{y})$ iff $\Gamma' \models \varphi_f$. Indeed, Γ' simply adds $\bar{y}$ to the propagated variables in each service. Therefore, we only consider in the rest of the chapter quantifier-free LTL-FO formulas.

4.3 The Spin-based Verifier

In this section we describe the implementation of SpinArt. The implementation is based on Spin, the widely used model checker in software verification. A brief review of Spin and *Promela*, the specification language for Spin, is provided in Appendix 4.6.

Building an artifact verifier based on Spin is a challenging task due to limitations of Spin and Promela. In Promela, one can only specify variables with bounded domains (**byte**, **int**, etc.) and bounded size (i.e. arrays with dynamic allocation are not allowed), but in the TAS model, the domains of the artifact variables and the database are unbounded and the database instance can have arbitrary size, so a direct translation is not possible. In addition, Spin cannot handle Promela programs of large size because the generated verifier V would be too large for the C compiler. Spin could also fail due to space explosion in the course of verification. Thus, our implementation requires a set of nontrivial translations and optimizations, discussed next.

4.3.1 Symbolic Verification

The implementation makes use of the symbolic representation technique developed in Chapter 3 to establish decidability and complexity results for HAS. With the symbolic representation, the verification of TAS's is reduced to finite-state model checking that Spin can handle. Intuitively, given a TAS specification Γ and an LTL-FO property φ_f, we use isomorphism types to describe symbolically the structure of the portion of the database reachable from the current tuple of artifact variables by navigating the foreign keys. An isomorphism type fully captures the information needed to evaluate any condition in Γ and φ_f. In addition, we can show, similarly to HAS, that to check whether $\Gamma \models \varphi_f$, it is sufficient to check that all *symbolic runs* of isomorphism types satisfy φ_f, or equivalently, that no symbolic run satisfies $\neg\varphi_f$. We define symbolic runs next.

We start with defining *isomorphism types* that are similar to the T-isomorphism types (Definition 31) for HAS. Essentially, an isomorphism type τ consists of a set of expressions $\mathcal{E}$ and an equality type $\sim_\tau$ of the expressions. For a set of variables $\bar{y}$, we denote by $\mathcal{E}(\bar{y})$ the set of symbols containing all variables $\bar{y}$, navigations starting from $\bar{y}$, and the set of all constants that appear in Γ or φ_f. Note that the length of each expression is bounded because of acyclicity of the foreign keys, so $\mathcal{E}$ is a finite set. We also include a tuple of variables $\bar{y}$ in the definition for convenience in the technical development in this chapter. We can now define isomorphism types.

Definition 105. *Let Γ be a TAS with variables $\bar{x}$, and φ_f an LTL-FO property of Γ. An isomorphism type τ for Γ, φ_f, and variables $\bar{y} \subseteq \bar{x}$ consists of a navigation set $\mathcal{E}(\bar{y})$ together with an equivalence relation $\sim_\tau$ over $\mathcal{E}(\bar{y})$ such that:*

- *$c \not\sim_\tau c'$ for constants $c \neq c'$ in $\texttt{const}(\Gamma, \varphi_f)$, and*
- *if $u \sim_\tau v$ and $u.f, v.f \in \mathcal{E}(\bar{y})$ then $u.f \sim_\tau v.f$.*

Example 106. *Figure 4.4 shows an isomorphism types τ of variables $\{x, y, z\}$, where $R(\text{ID}, A)$ is the only database relation, $\{x, y, z\}$ are 3 variables of type $R.\text{ID}$ and there is only one non-ID* *constant c_0. Each pair of expressions (e, e') are connected with an solid line ($=$-edge) if $e \sim_\tau e'$ otherwise a dashed line ($\neq$-edge). The $\neq$-edges between $\{x, y, z\}$ and $\{x.A, y.A, z.A, c_0\}$ are omitted in the figure for clarity. Note that since (x, y) is connected with an $=$-edge, $(x.A, y.A)$ must also be connected with $=$-edge as enforced by the key dependency.*

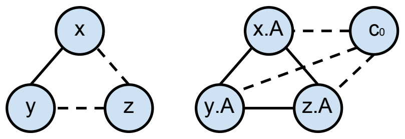


Figure 4.4. An isomorphism type of variables $\{x, y, z\}$.

Note that when $\bar{y} = \bar{x}$, τ provides enough information to evaluate conditions over $\bar{x}$. Satisfaction of a condition of φ_f by an isomorphism type τ is defined analogously to satisfaction by a T-isomorphism type, defined in Chapter 3.

Let τ be an isomorphism type with navigation set $\mathcal{E}(\bar{y})$ and equality type $\sim_\tau$. The projection of τ onto a subset of variables $\bar{z}$ of $\bar{y}$, denoted as $\tau|\bar{z}$, is $(\sim_\tau |\bar{z}, \mathcal{E}(\bar{z}))$ where $\sim_\tau |\bar{z}$ is the projection of $\sim_\tau$ onto $\mathcal{E}(\bar{z})$. We define the symbolic transition relation among isomorphism types as follows: for a TAS service $\sigma = (\pi, \psi, \bar{y})$ in Σ, $\tau \xrightarrow{\sigma} \tau'$ iff $\tau \models \pi$, $\tau' \models \psi$ and $\tau|\bar{y} = \tau'|\bar{y}$.

Definition 107. *A symbolic run of $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ is a sequence $\tilde{\rho} = \{(\tau_i, \sigma_i)\}_{i \geq 0}$ such that for each $i \geq 0$, τ_i is an isomorphism type, $\sigma_i \in \Sigma$, $\sigma_0 = \text{init}$, $\tau_0 \models \Pi$ and $\tau_i \xrightarrow{\sigma_{i+1}} \tau_{i+1}$.*

Example 108. *Figure 4.5 shows an example of applying a symbolic transition on an isomorphism type. The previous isomorphism type τ (top-left) satisfies the pre-condition, the next isomorphism type τ' (bottom) satisfies the post-condition, and they are consistent in their projection to the propagated variables $\{x, z\}$ (top-right).*

Satisfaction of a quantifier-free LTL-FO property on a symbolic run is defined in the standard way. One can show the following.

Theorem 109. *Given a TAS Γ and LTL-FO property φ_f of Γ, $\Gamma \models \varphi_f$ iff for every symbolic run $\tilde{\rho}$ of Γ, $\tilde{\rho} \models \varphi_f$.*

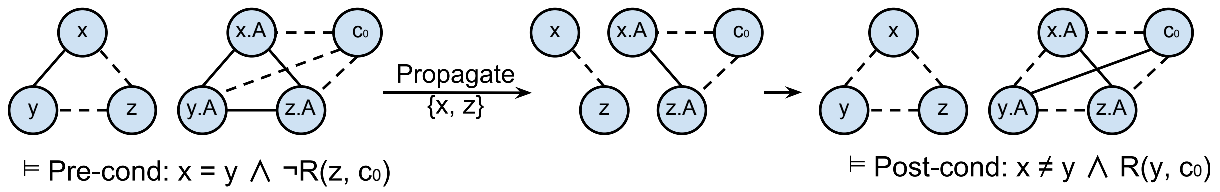


Figure 4.5. Symbolic transition.

4.3.2 Implementation of SpinArt

Using Theorem 109, one can implement a verifier that constructs a Promela program $\mathcal{P}$ to simulate the non-deterministic execution of symbolic transitions. The program $\mathcal{P}$ specifies $\mathcal{E}(\bar{x})$ as its variables. Each condition ψ in Γ and φ_f is translated into a Promela condition $f(\psi)$ as follows.

- if $\psi = (x = y)$, then $f(\psi) = \psi$;
- if $\psi = R(x, y_1, \ldots, y_m)$ for relation $R(\textbf{\textit{ID}}, A_1, \ldots, A_m)$, then $f(\psi) = \bigwedge_{i=1}^{m}(x.A_i = y_i)$;
- Boolean connectives are handled in the standard way.

Then $\mathcal{P}$ simulates the following process of executing symbolic transitions. First, $\mathcal{P}$ initializes the constant expressions with distinct values and other expressions with non-deterministically chosen values that satisfy $f(\Pi)$. Then for each service $\sigma = (\pi, \psi, \bar{y})$, we construct a non-deterministic option with guard $f(\pi)$ that executes the following:

(i) For each expression $e \in \mathcal{E}(\bar{x}) - \mathcal{E}(\bar{y})$, assign to e a non-deterministically chosen value from $\{0, \ldots, |\mathcal{E}(\bar{x})| - 1\}$.

(ii) Proceeds if $f(\psi)$ is `True` and for each pair of expressions e and e', $e = e'$ implies that for every attribute A where $\{e.A, e'.A\} \subseteq \mathcal{E}(\bar{x})$, $e.A = e'.A$. Otherwise the run is blocked and invalidated.

Example 110. *First, each TAS condition is translated into a condition in Promela. For example, the pre-condition in Example 108 is translated into*

```
(x == y) && !(z.A == c0).
```

Then, to construct the Promela program $\mathcal{P}$, we first have a ***do****-statement to ensure that the options constructed according to Section 4.3.2 are repeatedly chosen non-deterministically and executed. For example, the service in Example 108 is translated into the fragment of a Promela program shown in Fig. 4.6, where the* `select(y : 0 .. N - 1)` *statement is a built-in macro for assigning a variable with a value non-deterministically chosen from a range (here N is a constant equal to $|\mathcal{E}(\bar{x})|$).*

```
do
// check the pre-condition
:: ((x == y) && !(z.A == c0)) ->
   // choose values for y and y.A non-deterministically
   select(y : 0 .. N - 1);
   select(y.A : 0 .. N - 1);
   // validate the post-condition
   if
   :: (x != y && y.A == c0) -> skip;
   fi;
   // validate the Keys and FKs
   if
   :: ((x != y || x.A == y.A) && (y != z || y.A == z.A) && (
      x != z || x.A == z.A)) -> skip;
   fi;
:: // another service
...
od
```

Figure 4.6. A fragment of a Promela program translated from a service.

Intuitively, each valid valuation v to $\mathcal{E}(\bar{x})$ corresponds to a valid isomorphism type τ of $\bar{x}$ where $e \sim_\tau e'$ iff $v(e) = v(e')$. The guard ensures that the pre-condition holds. Part (i) ensures that the set of next valuations covers all possible valid successors of isomorphism types. Finally, the conditions in (ii) ensure that the post-condition holds and the keys and FKs dependencies are satisfied in the next isomorphism type.

Finally, the LTL-FO formula φ_f is translated into a LTL formula $\tilde{\varphi_f}$ in Promela by replacing each FO component c with $f(c)$ defined above. The universally quantified variables of

φ_f are translated into extra variables added to the Promela program. Small modifications to the LTL formula are also needed to skip the internal steps for assigning values and testing conditions in the run such that the Spin verification only considers the snapshots right after complete service applications. We can show the following.

Lemma 111. *Every symbolic run $\tilde{\rho} = \{(\tau_i, \sigma_i)\}_{i \geq 0}$ satisfies φ_f iff $\mathcal{P} \models \tilde{\varphi_f}$.*

The intuition of the above Lemma is that each valid valuation v to $\mathcal{E}(\bar{x})$ in $\mathcal{P}$ corresponds to an unique isomorphism type τ. The translated transitions in Promela guarantees that the set of runs of $\mathcal{P}$ captures the set of all symbolic runs. So to check whether Γ satisfies φ_f, it is sufficient to translate (Γ, φ_f) into $(\mathcal{P}, \tilde{\varphi_f})$ and verify whether $\mathcal{P} \models \tilde{\varphi_f}$.

However, this approach is inefficient in practice for the following reasons. In part (ii), the size of the tests to ensure satisfaction of the key and foreign key dependencies is quadratic in the number of expressions, so the compilation of $\mathcal{P}$ and the generated verifier is slow or simply fails. In (i), assigning to each e values from $\{0, \ldots, |\mathcal{E}(\bar{x})| - 1\}$ is also infeasible because it leads to state explosion when the actual search is performed by the verifier. As shown by the experiments, this leads to either slow execution or memory overflow. To overcome these two major obstacles, we introduce two key optimizations.

4.3.3 Optimization with Lazy Dependency Tests

In the first optimization, we reduce the size of the generated Promela program by eliminating the tests of key and foreign key dependencies in step (ii) of the above approach. Instead, we introduce tests of the dependencies in a lazy manner, only when two expressions are actually tested for equality. Formally, instead of performing the tests in (ii), we translate each condition ψ of (Γ, φ_f) into $f(\psi)$ then add the following additional tests: for every atom $(e = e')$ in the negation normal form[1] of $f(\psi)$, we replace $(e = e')$ with $\left(\bigwedge_{w:\{e.w, e'.w\} \subseteq \mathcal{E}(\bar{x})} e.w = e'.w\right)$ where w is a sequence of attributes.

[1] With negations pushed down and merged with the $=$ and $\neq$ atoms, the only remaining Boolean operators are $\wedge$ and $\vee$.

The size of the tests in the resulting Promela program $\mathcal{P}$ is $O((|\pi| + |\psi|) \cdot \max_{x \in \bar{x}} |\mathcal{E}(x)|)$ for each service, while the original size is $O(|\mathcal{E}(\bar{x})|^2 \cdot a)$ where a is the maximum arity in the database schema $\mathcal{DB}$. Typically, the size of a condition is much smaller than the number of expressions and $\max_{x \in \bar{x}} |\mathcal{E}(x)|$ is also smaller than $|\mathcal{E}(\bar{x})|$. We can see that the lazy dependency significantly reduces the size of the tests.

Example 112. *Consider the database schema $\mathcal{DB} = \{R(\text{ID}, A, B), S(\text{ID}, C, D)\}$ where A and B are foreign key attributes referencing the* ID *of S and C, D are non-key attributes. A condition $R(x, y, z)$ is translated into* `(x.A == y && x.B == z)` *without the optimization and* `(x.A == y && x.B == z && x.A.C == y.C && x.A.D == y.D && x.B.C == y.C && x.B.D == y.D)` *if the lazy dependency tests optimization is applied. The additional terms in the conditions are added so that the tests for keys and FKs in the translation can be removed.*

Example 113. *Consider the service and the translation shown in Fig. 4.6. With lazy dependency tests, the translated pre-condition becomes* `(x == y) && !(z.A == c0) && (x.A == y.A)` *with one additional term* `(x.A == y.A)`. *The translated post-condition is unchanged and the tests for keys and FKs are removed (lines 12-14). The overall size of the translation is reduced.*

Correctness. The modified translation using lazy dependency tests preserves correctness. The intuition is the following. With the lazy tests, in some snapshot with valuation v in the execution of $\mathcal{P}$, there could be two expressions e, e' where $v(e) = v(e')$ and for some attribute A, $v(e.A) \neq v(e'.A)$, but this does not matter because $e = e'$ is never tested during the current lifespan of e and e' (the segment of the symbolic run where e and e' are propagated), and neither are any of the prefixes of e and e'. So within the same lifespan, we are free to replace $v(e)$ and $v(e')$ with different values and the run of $\mathcal{P}$ remains valid. Thus, there is no need to enforce the equality $e.A = e'.A$.

4.3.4 Optimization by Assignment Set Minimization

In the naive approach, assigning expressions with values chosen from a set of size $|\mathcal{E}(\bar{x})|$ guarantees correctness by covering all possible isomorphism types, but it results in a large search space for Spin, which can lead to poor performance or memory overflow. The goal of this optimization is to reduce the size of the search space by minimizing the set of values used in the assignments while preserving the correctness of verification.

We denote by $A(e)$ the *assignment set* of a non-constant expression e, which is the set from which the Promela program $\mathcal{P}$ chooses non-deterministically values for e. The technique relies on static analysis of $\mathcal{P}$ and the translated property $\tilde{\varphi_f}$, aiming to reduce the size of the assignment sets as much as possible.

The intuition behind the optimization is the following. We notice that searching for an accepting run in the generated Promela program $\mathcal{P}$ can be regarded as searching for a sequence of sets of constraints $\{C_i\}_{i\geq 0}$, where each C_i consists of the (in)equality constraints imposed on the current snapshot by the history of the run. More precisely, the statements executed in $\mathcal{P}$ can be divided into two classes: (1) testing a condition π and (2) assigning new values to some expressions. At snapshot i, executing an (1)-statement can be viewed as adding π to C_i while C_i should remain consistent (no contradiction implied by the $=$-or-$\neq$ constraints in C_i), and a (2)-statement assigning a value to e can be viewed as projecting away from C_i constraints that involve e. When we construct the assignment set $A(\cdot)$, it is sufficient for correctness that the valuations generated with $A(\cdot)$ can witness the set of all reachable C_i's, which can be a small subset of all the possible isomorphism types. Thus, the resulting $A(\cdot)$ can be much smaller.

Computing all reachable C_i's can be as hard as the verification problem itself. So instead, we over-approximate them with the *constraint graph* G of $(\mathcal{P}, \tilde{\varphi_f})$ obtained by collecting all (in)equalities from $(\mathcal{P}, \tilde{\varphi_f})$, so that all C_i's are subgraphs of G.

Formally, the constraint graph G is an undirected labeled graph with $\mathcal{E}(\bar{x})$ as the set of nodes, where an edge $(e, e', \circ)$ is in G for $\circ \in \{=, \neq\}$ if $(e \circ e')$ is an atom in any condition of $\mathcal{P}$

and $\tilde{\varphi_f}$ with all conditions converted in negation normal form.

A subgraph G' of G is *consistent* if its edges do not lead to a contradiction (i.e., two nodes connected in G' by a sequence of $=$-edges are not also connected by an $\neq$-edge). Observe that G itself is generally not consistent, since it may contain mutually exclusive constraints that never arise in the same configuration. On the other hand, each C_i as above corresponds to a consistent subgraph of G.

Intuitively, the approach to minimizing the assignment sets proceeds as follows. First, consider the connected components of G with respect to its equality edges. Clearly, distinct connected components can be consistently assigned disjoint sets of values. Next, within each connected component, all expressions can be provided with the same assignment set, which we wish to minimize subject to the requirement that it must provide sufficiently many values to satisfy each of its consistent subgraphs.

More precisely, we can show the following.

Lemma 114. *Let $\mathcal{P}'$ be the Promela program obtained from $(\mathcal{P}, \tilde{\varphi_f})$ by replacing the assignment sets with any $A(\cdot)$ that satisfies:*

1. *for every $(e, e', =) \in G$, $A(e) = A(e')$, and*
2. *for every consistent subgraph G' of G, there exists a valuation v such that for every $e \in \mathcal{E}(\bar{x})$, $v(e) \in A(e)$ and for $\circ \in \{=, \neq\}$, $v(e) \circ v(e')$ if $(e, e', \circ) \in G'$.*

Then $\mathcal{P} \models \tilde{\varphi_f}$ iff $\mathcal{P}' \models \tilde{\varphi_f}$.

Note that constants are not taken into account in the above lemma but can be included in a straightforward way. Condition 2 implies that whenever a new valuation v' is generated from a previous valuation v, regardless of the previous and next constraint sets C and C', there exists a v' that is consistent with v, C and C'.

We next consider minimizing the assignment sets within each connected component. It turns out that computing the minimal $A(\cdot)$ that satisfies the above conditions is closely related to computing the *chromatic number* of a graph [59]. Recall that the chromatic number $\chi(G)$ of an

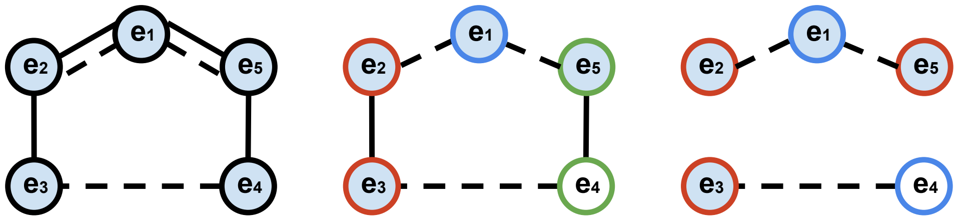


Figure 4.7. Example of Assignment Sets Minimization.

undirected graph G is the smallest number of colors needed to color G such that no two adjacent nodes share the same color. If the subgraph G' in condition 2 is fixed, then the minimal $|A(\cdot)|$ is precisely the chromatic number of G' restricted to only $\neq$-edges and with connected components of the $=$-edges merged into single nodes. We illustrate it with an example.

Example 115. *Consider the constraint graph G in the left of Fig. 4.7. The solid lines represent $=$-edges and the dashed lines represent $\neq$-edges. The entire graph consists of a single connected component of $=$-edges. To find the minimal $A(\cdot)$, we need to find the largest chromatic number over all consistent subgraphs of G. Consider two consistent subgraphs G_1 (middle) and G_2 (right). The chromatic number of G_1 is 3 because (e_2, e_3) (and (e_4, e_5)) must share the same color, so G_1 is in fact a triangle. The chromatic number of G_2 is 2 as it no long requires e_2 and e_5 to have different colors. In fact, G_1 is the subgraph with the largest chromatic number, so setting $A(e_i) = \{0, 1, 2\}$ for every i minimizes the assignment sets.*

As computing the chromatic number is NP-HARD, it is not difficult to show that computing $A(\cdot)$ with minimal size is also NP-HARD. (We conjecture that it is Π_2^{P}-HARD.) So computing the minimal $A(\cdot)$ can be inefficient. In the implementation, we use a simple algorithm that approximates the maximal chromatic number with the straightforward bound $\chi(G)(\chi(G)-1) \leq 2m$ where m is the number of $\neq$-edges within the connected component. The algorithm ensures satisfaction of the two conditions and produces reasonably small assignment sets in practice because the constraint graph is likely to be very sparse and contains few $\neq$-edges. This is confirmed by our experiments.

Table 4.1. Statistics of the BPMN benchmark.

#Workflows	Avg(#Relations)	Avg(#Variables)	Avg(#Services)
32	3.563	20.63	11.59

4.4 Experimental Results

In this section we describe the experiments evaluating the performance of SpinArt.

Benchmark. The benchmark used for the experiments consists of a collection of 32 artifact systems modeling realistic business processes from different application domains. Because of the difficulty in obtaining fully specified real-world data-driven business processes, we constructed the benchmark starting from business processes specified in the widely used BPMN model, that are provided by the official BPMN website [1]. We rewrote the BPMN specifications into artifact systems by manually adding the database schema, variables and service pre-and-post conditions. Table 4.1 provides some characteristics of the benchmark. The full benchmark has more features and is discussed in more detail in Section 5.4 of Chapter 5.

LTL-FO Properties. On each workflow in the benchmark, we run SpinArt on a collection of 12 LTL-FO properties constructed using templates of real propositional LTL properties, yielding a total of 384 runs. The LTL properties are all the 11 examples of safety, liveness and fairness properties collected from a standard reference paper [116] and an additional property `False` used as a baseline when comparing the performance of SpinArt on different classes of LTL-FO properties. We list all the templates of LTL properties in Table 4.3. We choose `False` as a baseline because it is the simplest property verifiable by Spin. By comparing the running time for a property with the running time for `False` on the same specification, we obtain the overhead for verifying the property.

For each workflow, we generate an LTL-FO property corresponding to each template by replacing the propositions with FO conditions chosen from the pre-and-post conditions of all the services and their sub-formulas. Note that by doing so, the generated LTL-FO properties on the

Table 4.2. Performance of SpinArt in different modes.

Mode	#Failed-Runs	Total-Time	Verify-Time	Compile-Time	#States
SpinArt-NoASM	48 / 384	21.399s	14.379s	7.020s	1,547,211
SpinArt-NoLDT	3 / 384	12.240s	3.769s	8.471s	809,025
SpinArt-Full	**3** / 384	**2.970s**	**0.292s**	**2.678s**	**44,826**

real workflows are combinations of real propositional LTL properties and real FO conditions, and so are close to real-world LTL-FO properties.

Setup. We implemented SpinArt in `C++` with Spin version 6.4.6. All experiments were performed on a Linux server with a quad-core Intel i7-2600 CPU and 16G memory. To allow larger search space, Spin was run with the state compression optimization turned on. For faster execution, the Spin-generated verifier was compiled with `gcc` and the -O2 optimization. The time and memory limit of each run was set to 10 minutes and 8G respectively.

Performance. In addition to running the full verifier (SpinArt-Full), we also ran the verifier with the lazy dependency tests optimization (LDT) turned off (SpinArt-NoLDT) and with assignment set minimization (ASM) turned off (SpinArt-NoASM). For all the verifiers, we compare their number of failed runs (timeout or memory overflow), the average compilation time[2] for generating the executable verifier (Compile-Time), the average execution time of the generated verifier (Verify-Time), the average total running time (Verify-Time + Compile-Time), and the average number of reached states as reported by Spin.

The results are shown in Table 4.2. We can see that the performance of SpinArt is promising. Its average total running time is within 3 seconds and there are only 3/384 failed runs ($<1\%$) due to memory overflow. This is a strong indication that the approach is sufficiently practical for real-world workloads. The full verifier is also significantly improved compared to SpinArt-NoLDT and SpinArt-NoASM. Without ASM, the the verifier failed on 12.5% (48/384) of all runs and the average running time is >7x times faster when the optimization is turned on. Without LDT, most of the runs are still successful, but the average total running time is >4 times

[2]All averages (running times and #States) are taken over the successful runs.

faster with the optimization turned on. Both optimizations significantly reduce the size of the state space ($>$95% in total), resulting in much shorter verification time.

We next discuss the effect of each optimization in more detail.

Effect of Lazy Dependency Tests. From Table 4.2, we observe that for the successful runs, compilation time accounts for a large fraction of the total running time, so minimizing the size of the Promela program is critical to improve the overall performance of a Spin-based verifier. Figure 4.8 shows the changes in the compilation time as the size of the input specification (#Variables + #Services) increases, for runs with or without the LDT optimization. Each point in the figure corresponds to one specification and the compilation time is measured by the average compilation time of all runs of the specification. The figure shows that with LDT, the compilation time grows not only slower as the input size increases, but in some cases it can compile $>$10 times faster than compilation without LDT. Overall, LDT leads to an average speedup of 3.2x in compilation.

Effect of Assignment Set Minimization. We show the effectiveness of Assignment Set Minimization (ASM) by comparing the approximation algorithm for ASM with a naïve approach (NoASM) where the size of the assignment set of each expression e is simply set to the number of expressions having the same type as e. Figure 4.9 shows the growth of the average size of the assignment sets as the size of the input specification increases. For ASM, the average size stays very low (2.05 in average) as the input size grows. This shows that our algorithm is near-optimal in practice. Compared to the naive approach where the average size increases linearly with the input size, our approach produces much smaller assignment sets. In some cases, the assignment set generated by the algorithm is $>$30 times smaller than the ones generated by the naive approach.

Effect of the Structure of LTL-FO Properties. Next, we measure the performance on different classes of LTL-FO properties. Table 4.3 lists all the LTL templates used in generating the LTL-FO properties and their intuitive meaning, as in [116]. For each template, we measure

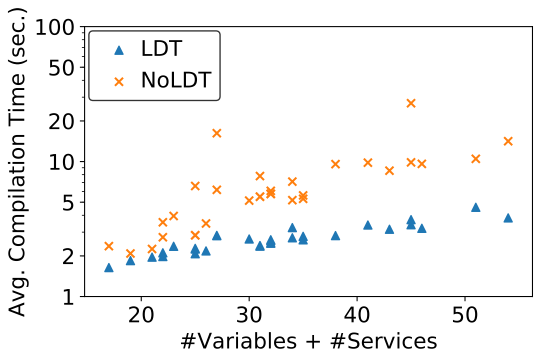


Figure 4.8. Compilation time with or without Lazy Dependency Tests.

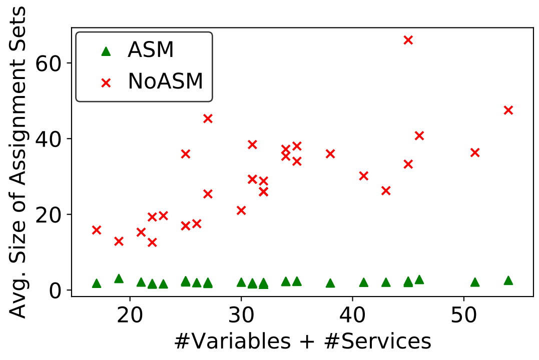


Figure 4.9. Average size of the assignment sets with or without minimization.

Table 4.3. Average running time of verifying different classes of LTL-FO properties.

Templates	Avg(Time)	Overhead	Templates	Avg(Time)	Overhead
`False`	2.68s	0.00%	$\mathbf{G}(\varphi \rightarrow \mathbf{F}\psi)$	2.72s	1.45%
$\mathbf{G}\varphi$	2.68s	-0.26%	$\mathbf{F}\varphi$	2.80s	4.08%
$(\neg\varphi\ \mathbf{U}\ \psi)$	2.70s	0.61%	$\mathbf{GF}\varphi \rightarrow \mathbf{GF}\psi$	2.91s	9.36%
$(\neg\varphi\mathbf{U}\psi) \wedge \mathbf{G}(\varphi \rightarrow \mathbf{X}(\neg\varphi\mathbf{U}\psi))$	5.07s	70.02%	$\mathbf{GF}\varphi$	3.07s	15.14%
$\mathbf{G}(\varphi \rightarrow (\psi \vee \mathbf{X}\psi \vee \mathbf{XX}\psi))$	2.72s	1.40%	$\mathbf{G}(\varphi \vee \mathbf{G}\psi)$	2.71s	0.85%
$\mathbf{G}(\varphi \vee \mathbf{G}(\neg\varphi))$	2.69s	0.28%	$\mathbf{FG}\varphi \rightarrow \mathbf{GF}\psi$	2.91s	9.11%

the average running time over all runs with LTL-FO properties generated using the template. In addition, we measure the overhead of verifying a LTL-FO property by comparing with its running time for the property `False`, the simplest non-trivial property for SpinArt. The overhead of a class of LTL-FO properties is obtained by the average overhead of all properties of the same class. The result in Table 4.3 shows that the average running time stays within 2x of the average running time for `False` and the maximum average overhead is about 70%. The overhead increases as the LTL property becomes more complex, but is within a reasonable range. Note that this is much better than the theoretical upper bound, which is exponential in the size of the LTL formula.

4.5 Conclusion and Discussion

We described in this chapter the implementation of SpinArt, a verifier for data-driven workflows using the widely used off-the-shelf model checker Spin. With a translation based on the symbolic representation developed in Chapter 3 enhanced with nontrivial optimizations, SpinArt achieves good performance on a realistic business process benchmark. We believe this is a first successful attempt to bridge the gap between theory and practice in verification of data-driven workflows, with full support for unbounded data and relying on an off-the-shelf model checker.

The focus of this chapter is on sound and complete artifact verifiers, in contrast to incomplete verifiers (e.g. based on theorem provers). Within this scope, SpinArt establishes a practical trade-off point on the spectrum ranging from using off-the-shelf general software verifiers to developing dedicated verifiers from scratch.

However, off-the-shelf tools share a number of limitations which are inherited by verifiers based on them (including SpinArt). For instance, general-purpose model checkers have limited support for unbounded data. While this chapter mitigates this limitation by supporting the unbounded read-only database with symbolic representation, our model does not support other ingredients of the HAS (and GSM) model, such as dynamically updatable artifact relations, because as shown in Chapter 3 they require an enhanced symbolic representation based on VASS, counting the number of tuples of different isomorphism types, which exceeds the capabilities of Promela/Spin. Moreover, the encoding of the symbolic representation also hurts the performance since some of the most effective optimizations cannot be encoded in languages like Promela. These disadvantages indicate the need for a specialized implementation, which we present in the next chapter.

```
if
:: (a == 0) -> b = a + 1;
:: (b > 1) -> c = a;
:: a = a - 1;
   b = b + 1;
fi
```

```
do
:: count = count - 1;
:: a = a + 2;
:: (count == 0) -> break;
:: (count > 0) -> skip;
od
```

Figure 4.10. Examples of Promela program (Left: **if**-statement; Right: **do**-statement).

4.6 Appendix: Review of Spin and Promela

The implementation of our artifact verifier relies on Spin, a widely used model checker in software verification. Spin supports the verification of LTL properties of models specified in *Promela*, a C-like modeling language for parallel systems. At a high level, a single-process Promela program can be viewed as a non-deterministic C program, where one can specify variables of fixed bit-length (e.g. byte, short, int) and statements that manipulate the variables (e.g. assignments, goto, etc.). Non-determinism is specified using the **if**- and **do**-statements illustrated in Fig. 4.10.

When the **if**-statement is executed, one of its options with no guard or with its guard evaluating to `True` is chosen non-deterministically and executed. Each option is a sequence of one or more statements. If no option can be chosen, then the run blocks the is not considered as a valid run when Spin is executed. The **do**-statement is similar to the **if**-statement, with the difference that the execution is repeated after an option is completed. Nesting is allowed within the **if**- or **do**-statements.

Developers can verify LTL properties of a Promela program using Spin. Given a Promela program $\mathcal{P}$, a developer can write LTL properties where the propositions are Boolean conditions over the variables of $\mathcal{P}$, such as: "`G ((a == 1) -> F (b > 0 || c < 0))`".

To check satisfaction of a LTL property φ, Spin first produces the source code of a problem-specific verifier V in C. Then V is compiled with a C-compiler (e.g. `gcc`) and executed to produce the result.

Acknowledgement

This chapter contains material from “SpinArt: A Spin-based Verifier for Artifact Systems” by Yuliang Li, Alin Deutsch and Victor Vianu. This paper is in ArXiv [88]. The dissertation author was the primary investigator of this paper.

Chapter 5

VERIFAS: A Practical Verifier for Artifact Systems

5.1 Overview

This chapter presents VERIFAS, an artifact verifier built from scratch relaying on the verification algorithm developed in Chapter 3. The main contributions are the following.

- We define HAS*, a variant of HAS which strikes a more practically relevant trade-off between expressivity and verification complexity, as demonstrated by its ability to specify a realistic set of business processes. We adapt to HAS* the theory developed for HAS in Chapter 3, laying the groundwork for our implementation.
- We implement VERIFAS, a fully automatic verifier for HAS*. The implementation makes crucial use of *novel optimization techniques*, with dramatic impact on performance. The optimizations are non-trivial and include concise symbolic representations, aggressive pruning in the search algorithm, and the use of highly efficient data structures.
- We evaluate the performance of VERIFAS using both real-world and synthetic artifact systems and properties from a benchmark we create, bootstrapping from existing sets of business process specifications and properties by extending them with data-aware features. To our knowledge, this is the first benchmark for business processes and properties that includes such data aware features. The experiments highlight the impact of the various optimizations and

parameters of both the artifact systems and properties.

- We adapt to HAS* a standard complexity measure of control flow used in software engineering, *cyclomatic complexity* [128], and show experimentally, using the above benchmark, that cyclomatic complexity of HAS* specifications correlates meaningfully with verification times. Since conventional wisdom in software engineering holds that well-designed, human readable programs have relatively low cyclomatic complexity, this is an indication that verification times are likely to be good for well-designed HAS* specifications.

Taking this and other factors into account, the experimental results show that our verifier performs very well on practically relevant classes of artifact systems. Compared to SpinArt, it not only applies to a much broader class of artifacts but also has a decisive performance advantage even on the simple artifacts that SpinArt is able to handle. To the best of our knowledge, this is the first implementation of practical significance of an artifact verifier with full support for unbounded data.

The rest of this chapter is organized as follows. We start by introducing in Section 5.2 the HAS* model supported by VERIFAS. Section 5.3 describes the implementation of VERIFAS by first reviewing in brief the theory developed in Chapter 3, particularly the symbolic representation technique used in establishing the theoretical results. We show an extension of the symbolic representation, called partial isomorphism type, to allow practical verification by adapting the classic Karp-Miller algorithm [83]. We then introduce three specialized optimizations to gain further performance improvement. We present our experimental results in Section 5.4. Finally, we conclude in Section 5.5. An appendix provides further technical details.

5.2 The Model

In this section we present the variant of Hierarchical Artifact Systems used in our study. The variant, denoted HAS*, differs from the HAS model used in Chapter 3 in two respects. On one hand, it *restricts* HAS as follows:

- it disallows arithmetic in service pre-and-post conditions
- the underlying database schema uses an *acyclic* set of foreign keys

On the other hand, HAS* *extends* HAS by removing various restrictions:

- tasks may have multiple updatable artifact relations
- each subtask of a given task may be called multiple times between task transitions
- certain restrictions on how variables are passed as parameters among tasks, or inserted/retrieved from artifact relations, are lifted

Because HAS* imposes some restrictions on HAS but removes others, it is incomparable to HAS. Intuitively, the choice of HAS* over HAS as a target for verification is motivated by the fact that HAS* achieves a more appealing trade-off between expressiveness and verification complexity. The acyclic schema restriction, satisfied by the widely used Star (or Snowflake) schemas [84, 126], is acceptable in return for the removal of various HAS restrictions limiting modeling capability. Indeed, as shown by our real-life examples, HAS* is powerful enough to model a wide variety of business processes. While the current version of VERIFAS does not handle arithmetic, the core verification algorithm can be augmented to include arithmetic along the lines developed for HAS. Limited use of aggregate functions can also be accommodated. These enhancements are left for future work.

We now present the syntax and semantics of HAS*. To avoid duplication, the definitions specify the new features and differences with the HAS model. These features are illustrated with an intuitive example of the HAS* specification of the order fulfillment business process shown in Chapter 4.

The database schemas of HAS* are acyclic database schemas, as defined for HAS. The running example contains the following database schema:

- $\texttt{CUSTOMERS}(\underline{\mathit{ID}}, \texttt{name}, \texttt{address}, \texttt{record})$
  $\texttt{ITEMS}(\underline{\mathit{ID}}, \texttt{item\_name}, \texttt{price})$

$\texttt{CREDIT\_RECORD}(\underline{\textit{ID}}, \texttt{status})$

In the schema, the IDs are key attributes, `price`, `item_name`, `name`, `address`, `status` are non-key attributes, and `record` is a foreign key attribute satisfying the dependency

$$\texttt{CUSTOMERS}[record] \subseteq \texttt{CREDIT\_RECORD}[\textit{ID}].$$

Intuitively, the `CUSTOMERS` table contains customer information with a foreign key pointing to the customers' credit records stored in `CREDIT_RECORD`. The `ITEMS` table contains information on the items. Note that the schema is acyclic as there is only one foreign key reference from `CUSTOMERS` to `CREDIT_RECORD`.

We next proceed with tasks and services for HAS*. A HAS* task is similar to a task in HAS with two new features: (1) a HAS* task T here can have a collection $\mathcal{S}^T$ of artifact relations and (2) each ID variable has a fixed type, meaning that its domain consists of the IDs of a fixed relation in the database $\mathcal{DB}$. Formally,

Definition 116. *A* ***HAS* task schema*** *over database schema $\mathcal{DB}$ is a tuple $T = \langle \bar{x}^T, \mathcal{S}^T, \bar{x}^T_{in}, \bar{x}^T_{out}\rangle$ where $\bar{x}^T$ is a sequence of artifact variables, $\mathcal{S}^T$ is a set of relation symbols not in $\mathcal{DB}$, and $\bar{x}^T_{in}$ and $\bar{x}^T_{out}$ are subsequences of $\bar{x}^T$. For each relation $\mathcal{S} \in \mathcal{S}^T$, we denote by $\texttt{attr}(\mathcal{S})$ the set of attributes of $\mathcal{S}$. The domain of each variable $x \in \bar{x}^T$ and each attribute $A \in \texttt{attr}(\mathcal{S})$ is either* $\text{DOM}_{val} \cup \{\texttt{null}\}$ *or $dom(R.ID) \cup \{\texttt{null}\}$ for some relation $R \in \mathcal{DB}$. In the latter case we say that the type of x (or A) is $\mathtt{type}(x) = R.ID$ ($\mathtt{type}(A) = R.ID$). An instance ρ of T is a tuple (ν, S) where ν is a valuation of $\bar{x}^T$ and S is an instance of $\mathcal{S}^T$ such that $S(\mathcal{S})$ is of the type of $\mathcal{S}$ for each $\mathcal{S} \in \mathcal{S}^T$.*

HAS* artifact schemas are defined analogously.

Definition 117. *A* ***HAS* artifact schema*** *is a tuple $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB}\rangle$ where $\mathcal{DB}$ is a HAS* database schema and $\mathcal{H}$ is a rooted tree of HAS* task schemas over $\mathcal{DB}$ with pairwise disjoint sets of artifact variables and distinct artifact relation symbols.*

Example 118. *The order fulfillment workflow has a task called* ***ProcessOrders****, which stores the order data and processes the orders by interacting with other tasks. It has the following artifact variables:*

- *ID variables:* `cust_id` *of type* `CUSTOMERS`.ID *and* `item_id` *of type* `ITEMS`.ID
- *non-ID variables:* `status` *and* `instock`

There are no input or output variables. The task also has an artifact relation `ORDERS`(`cust_id`, `item_id`, `status`, `instock`) *with attributes of the same types as the variables. Intuitively,* `ORDERS` *stores the orders to be processed, where each order consists of a customer and an ordered item. The variable* `status` *indicates the current status of the order and* `instock` *indicates whether the item is currently in stock.*

The order fulfillment workflow has 5 tasks: T_1: ***ProcessOrders****, T_2:****TakeOrder****, T_3:****CheckCredit****, T_4:* ***Restock*** *and T_5:****ShipItem****, which form the hierarchy represented in Figure 5.1. Intuitively, the root task* ***ProcessOrders*** *serves as a global coordinator which maintains a pool of all orders and the child tasks* ***TakeOrder****,* ***CheckCredit****,* ***Restock*** *and* ***ShipItem*** *implement the 4 sequential stages in the fulfillment of an order. At a high level,* ***ProcessOrders*** *repeatedly picks an order from its pool and processes it with a stage by calling the corresponding child task. After the child task returns, the order is either placed back into the pool or processed with the next stage. For each order, the workflow first obtains the customer and item information using the* ***TakeOrder*** *task. The credit record of the customer is checked by the* ***CheckCredit*** *task. If the record is good, then* ***ShipItem*** *can be called to ship the item to the customer. If the requested item is unavailable, then* ***Restock*** *must be called before* ***ShipItem*** *to procure the item.*

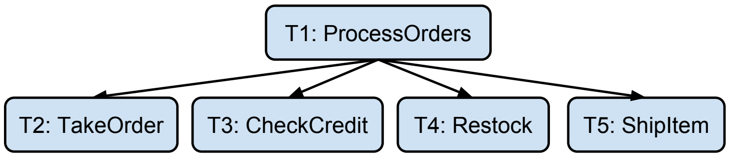


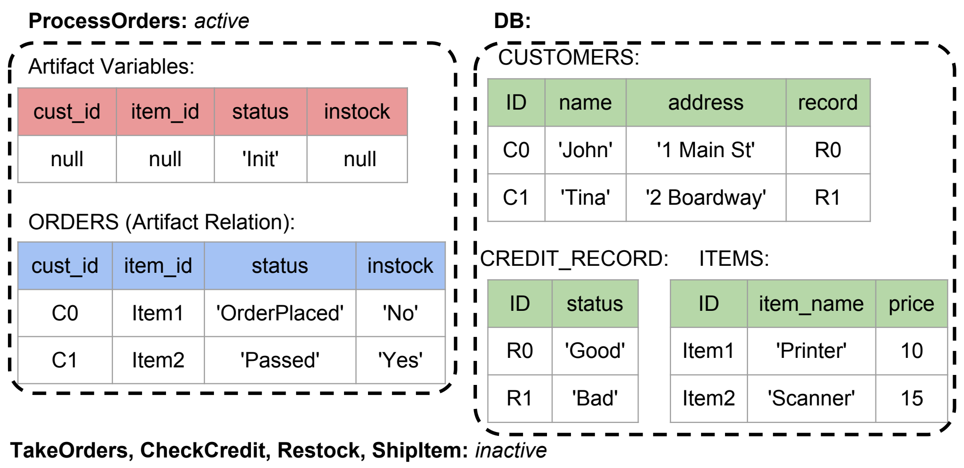
Figure 5.1. Tasks Hierarchy

Definition 119. *An* ***instance*** *of a HAS* artifact schema $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$ is a tuple $I = \langle \nu, stg, D, S \rangle$*

where D is a finite instance of $\mathcal{DB}$, S a finite instance of $\mathcal{S}_{\mathcal{H}}$, ν a valuation of $\bigcup_{i=1}^{k} \bar{x}^{T_i}$, and stg (standing for "stage") a mapping of $\{T_1, \ldots, T_k\}$ to $\{\texttt{active}, \texttt{inactive}\}$.

The stage $stg(T_i)$ of a task T_i has the following intuitive meaning in the context of a run of its parent: `active` says that T_i has been called and has not yet returned its answer, and `inactive` indicates that T_i is not active. Notice that the stage component $stg(T_i)$ of a task T_i takes only two values unlike in HAS. The stage `init` is no longer needed because a task T_i can be called any number of times within a given run of its parent, but only one instance of it can be active at any given time.

Example 120. *Figure 5.2 shows a partial example of an instance of the Order Fulfillment artifact system. The only active task is* ***ProcessOrder****.*



ProcessOrders: *active*

Artifact Variables:

cust_id	item_id	status	instock
null	null	'Init'	null

ORDERS (Artifact Relation):

cust_id	item_id	status	instock
C0	Item1	'OrderPlaced'	'No'
C1	Item2	'Passed'	'Yes'

DB:

CUSTOMERS:

ID	name	address	record
C0	'John'	'1 Main St'	R0
C1	'Tina'	'2 Boardway'	R1

CREDIT_RECORD:

ID	status
R0	'Good'
R1	'Bad'

ITEMS:

ID	item_name	price
Item1	'Printer'	10
Item2	'Scanner'	15

TakeOrders, CheckCredit, Restock, ShipItem: *inactive*

Figure 5.2. An instance of the Order Fulfillment workflow

We next define services of tasks. Like in HAS, each HAS* task contains internal services that update the artifact variables and artifact relation of the task.

Definition 121. *Let $T = \langle \bar{x}^T, \mathcal{S}^T, \bar{x}^T_{in}, \bar{x}^T_{out} \rangle$ be a task of a HAS* artifact schema $\mathcal{A}$. An* ***internal service*** *σ of T is a tuple $\langle \pi, \psi, \bar{y}, \delta \rangle$ where:*

- *π and ψ, called* pre-condition *and* post-condition*, respectively, are conditions over $\bar{x}^T$*
- *$\bar{y}$ is the set of* propagated variables*, where $\bar{x}^T_{in} \subseteq \bar{y} \subseteq \bar{x}^T$;*

- *δ, called the* update*, is a subset of* $\{+\mathcal{S}_i(\bar{z}), -\mathcal{S}_i(\bar{z}) | \mathcal{S}_i \in \mathcal{S}^T, \bar{z} \subseteq \bar{x}^T, \mathit{type}(\bar{z}) = \mathit{type}(\texttt{attr}(\mathcal{S}_i))\}$ *of size at most 1.*
- *if $\delta \neq \emptyset$ then $\bar{y} = \bar{x}^T_{in}$*

Like in HAS, an internal service σ of T can be called only when the current instance satisfies the pre-condition π. The update on variables $\bar{x}^T$ is valid if the next instance satisfies the post-condition ψ and the values of propagated variables $\bar{y}$ stay unchanged. This is a new feature compared to HAS, where the propagated variables are restricted to the input variables. Also, we note that a tuple of variables $\bar{z}$ is specified in each update of δ. These new features add flexibility in the manipulations of artifact variables and relations.

Any task variable that is not propagated can be changed arbitrarily during a task activation, as long as the post condition holds. This allows services to also model actions by external actors who provide input into the workflow by setting the value of non-propagated variables. Such actors may even include humans or other parties whose behavior is not deterministic. For example, a bank manager carrying out a “loan decision” action can be modeled by a service whose result is stored in a non-propagated variable and whose value is restricted by the post-condition to either “Approve” or “Deny”. Note that deterministic actors are modeled by simply using tighter post-conditions.

When $\delta = \{+\mathcal{S}_i(\bar{z})\}$, a tuple containing the *current* value of $\bar{z}$ is inserted into $\mathcal{S}_i$. When $\delta = \{-\mathcal{S}_i(\bar{z})\}$, a tuple is chosen and removed from $\mathcal{S}_i$ and the *next* value of $\bar{z}$ is assigned with the value of the tuple. Note that $\bar{x}^T_{in}$ are always propagated, and no other variables are propagated if $\delta \neq \emptyset$. The restriction on updates and variable propagation may at first appear mysterious. Its underlying motivation is that allowing simultaneous artifact relation updates and variable propagation turns out to raise difficulties for verification, while the real examples we have encountered do not require this capability.

Example 122. *The **ProcessOrders** task has 3 internal services:* Initialize*,* StoreOrder *and* RetrieveOrder*. Intuitively,* Initialize *creates a new order with* $\texttt{cust\_id} = \texttt{item\_id} = \texttt{null}$*. When* RetrieveOrder *is called, an order is chosen non-deterministically and removed from*

$\mathtt{ORDERS}$ *for processing, and* $(\mathtt{cust\_id}, \mathtt{item\_id}, \mathtt{status}, \mathtt{instock})$ *is set to be the chosen tuple. When* StoreOrder *is called, the current order* $(\mathtt{cust\_id}, \mathtt{item\_id}, \mathtt{status}, \mathtt{instock})$ *is inserted into* $\mathtt{ORDERS}$*. The latter two services are specified as follows.*

RetrieveOrder*:*

Pre: $\mathtt{cust\_id} = \mathtt{null} \wedge \mathtt{item\_id} = \mathtt{null}$

Post: $\mathtt{True}$

Update: $\{-\mathtt{ORDERS}(\mathtt{cust\_id}, \mathtt{item\_id}, \mathtt{status}, \mathtt{instock})\}$

StoreOrder*:*

Pre: $\mathtt{cust\_id} \neq \mathtt{null} \wedge \mathtt{item\_id} \neq \mathtt{null} \wedge \mathtt{status} \neq$ *"Failed"*

Post: $\mathtt{cust\_id} = \mathtt{null} \wedge \mathtt{item\_id} = \mathtt{null} \wedge \mathtt{status} =$ *"Init"*

Update: $\{+\mathtt{ORDERS}(\mathtt{cust\_id}, \mathtt{item\_id}, \mathtt{status}, \mathtt{instock})\}$

The sets of propagated variables are empty for both services.

An internal service of a HAS* task T specifies transitions that modify the variables $\bar{x}^T$ of T and the contents of $\mathcal{S}^T$. Figure 5.3 shows an example of a transition that results from applying the service *StoreOrder* of the **ProcessOrders** task.

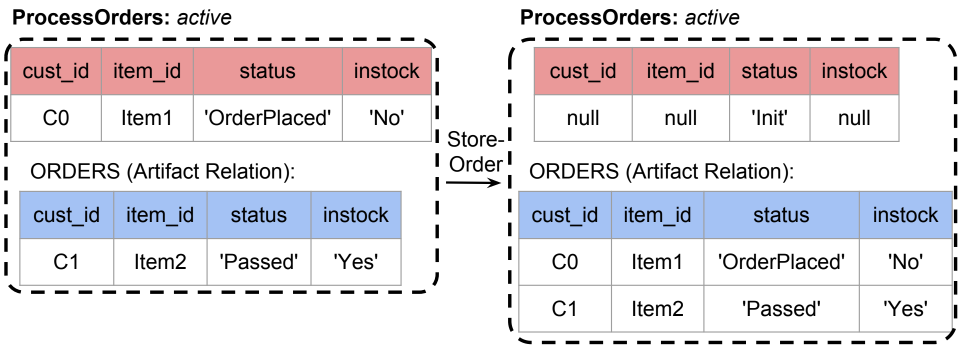


Figure 5.3. Transition caused by an internal service

As seen above, internal services of a task cause transitions on the data local to the task. Interactions among tasks are specified using the opening-services and closing-services already defined in HAS (Definition 14).

We are now ready to define HAS*.

Definition 123. *A* Hierarchical Artifact System* *(HAS*) is a triple* $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$*, where* $\mathcal{A}$ *is a HAS* artifact schema,* Σ *is a set of HAS* services over* $\mathcal{A}$ *including* σ_T^o *and* σ_T^c *for each task* T *of* $\mathcal{A}$*, and* Π *is a condition over* $\bar{x}^{T_1}$ *(the global pre-condition of* Γ*), where* T_1 *is the root task.*

We next define the semantics of HAS*. Two semantics are provided, the tree of local runs and the global runs, with relaxations to the HAS semantics allowing more flexibility. For clarity of presentation, we focus on modifications to the definition of local run. The full definitions can be obtained from those for HAS by taking the modifications into account. The differences lie in local transitions caused by internal services, and opening and closing of child tasks, which we define next.

A transition caused by an internal service is defined as follows.

Definition 124. *Let* $T = \langle \bar{x}^T, \mathcal{S}^T, \bar{x}^T_{\mathsf{in}}, \bar{x}^T_{\mathsf{out}} \rangle$ *be a HAS* task and* D *a database instance over* $\mathcal{DB}$*. An instance of* T *is a pair* (ν, S) *where* ν *is a valuation of* $\bar{x}^T$ *and* S *an instance of* $\mathcal{S}^T$*. For instances* $I = (\nu, S)$ *and* $I' = (\nu', S')$ *of* T *and an internal service* $\sigma = \langle \pi, \psi, \bar{y}, \delta \rangle$*, there is a local transition* $I \xrightarrow{\sigma} I'$ *if the following holds.*

- $D \models \pi(\nu)$ *and* $D \models \psi(\nu')$*,*
- $\nu'(\bar{y}) = \nu(\bar{y})$*,*
- *if* $\delta = \{+\mathcal{S}_i(\bar{z})\}$*, then* $S' = S[\mathcal{S}_i \mapsto S(\mathcal{S}_i) \cup \{\nu(\bar{z})\}]$*,*
- *if* $\delta = \{-\mathcal{S}_i(\bar{z})\}$*, then* $\nu'(\bar{z}) \in S(\mathcal{S}_i)$ *and* $S' = S[\mathcal{S}_i \mapsto S(\mathcal{S}_i) - \{\nu'(\bar{z})\}]$*,*
- *if* $\delta = \emptyset$ *then* $S' = S$*.*

The changes to the opening and closing of child tasks are simply the following:

- For every transition $I \xrightarrow{\sigma} I'$ where σ is a closing service $\sigma^c_{T_c}$, it is NOT required that $\nu'(z) = \nu(z)$ if z is an ID variable and $\nu(z) \neq \texttt{null}$. In other words, return values can overwrite arbitrary variables in HAS*
- Within each segment J of a local run ρ and for each child task T_c of T, there can be arbitrary number of $i \in J$ such that $\sigma_i = \sigma^o_T$. In addition, a child task T_c can be opened if $stg(T_c) =$

`inactive` (this is originally `init` in HAS). This means that a child task can be called multiple times within a segment (in between two internal service calls).

The semantics with the trees of local runs can be obtained by taking the above changes into account. Then the semantics with global runs can be obtained by following the same development as in HAS.

5.2.1 Specifying properties of HAS*

In this chapter we focus on verifying temporal properties of local runs of tasks in a HAS* using a variant of LTL-FO. For instance, in the order fulfillment example, we would like to specify in LTL-FO properties such as:

($\dagger$) If an order is taken and the ordered item is out of stock, then the item must be restocked before it is shipped.

LTL-FO for TAS was already defined in Chapter 4. The variant of LTL-FO for HAS* is slightly different as it applies to the *local* runs of a task T. The set of propositions is $P \cup \Sigma_T^{obs}$, where propositions in P are interpreted as conditions as before and Σ_T^{obs} consists of the services observable in local runs of T (including calls and returns from children tasks, see definition in Chapter 3).

We provide a flavor of the language using property ($\dagger$). The property is of the form $\varphi = \mathbf{G}(p \rightarrow (\neg q \ \mathbf{U}\ r))$, which means if p happens, then in what follows, q will not happen until r is true. Here p says that the **TakeOrder** task returned with an out-of-stock item, q states that the **ShipItem** task is called with the same item, and r states that the service **Restock** is called to restock the item. Since the item mentioned in p, q and r must be the same, the formula requires using a global variable i to record the item ID. This yields the following LTL-FO property:

$$\forall i\ \mathbf{G}((\sigma^c_{\texttt{TakeOrder}} \wedge \texttt{item\_id} = i \wedge \texttt{instock} = \text{“No”}) \rightarrow$$
$$(\neg(\sigma^o_{\texttt{ShipItem}} \wedge \texttt{item\_id} = i)\ \mathbf{U}\ (\sigma^o_{\texttt{Restock}} \wedge \texttt{item\_id} = i)))$$

A correct specification can enforce (†) simply by requiring in the pre-condition of $\sigma^o_{\texttt{ShipItem}}$ that the item is in stock. One such pre-condition is ($\texttt{instock} =$ "Yes" $\wedge$ $\texttt{status} =$ "Passed"), meaning that the item is in stock and the customer passed the credit check. However, in a similar specification where the $\texttt{instock} =$ "Yes" test is performed within **ShipItem** (i.e. in the pre-conditions of all shipping internal services) instead of the opening service of **ShipItem**, the LTL-FO property (†) is violated because **ShipItem** can be opened without first calling the **Restock** task. VERIFAS would detect this error and produce a counter-example illustrating the violation.

We use the standard definition for the satisfaction of a LTL-FO property $\forall \bar{y} \varphi_f$ by a local run ρ_T. Note that here we are interested in evaluating LTL-FO formulas $\forall \bar{y} \varphi_f$ on both infinite *and* finite local runs (infinite runs occur when a task runs forever).

Remark 125. *In Chapter 3, we consider HLTL-FO, a more complex logic for specifying properties of artifact systems. The presentation here focuses on verification of LTL-FO properties of individual tasks for simplicity. It can be shown that the basic techniques presented here for LTL-FO can be used as building block for verifying the more complex HLTL-FO properties. However, verifying LTL-FO properties of individual tasks is in fact adequate in most practical situations we have encountered.*

5.3 VERIFAS

In this section we describe the implementation of VERIFAS. We begin with a brief review of the theory developed in Chapter 3 that is relevant to the implementation.

5.3.1 Review of the Theory

The decidability and complexity results of HAS can be extended to HAS* by adapting the proofs and techniques developed there. We can show the following.

Theorem 126. *Given a HAS* Γ *and an LTL-FO formula* φ_f *for a task* T *in* Γ*, it is decidable in* EXPSPACE *whether* Γ *satisfies* φ_f*.*

We review informally the roadmap to verification developed for HAS, which is the starting point for the implementation. Let Γ be a HAS* and φ_f an LTL-FO formula for some task T of Γ. We would like to verify that every local run of T satisfies φ_f. Since there are generally infinitely many such local runs due so the unbounded data domain, and each run can be infinite, an exhaustive search is impossible. This problem is addressed in Chapter 3 by developing a symbolic representation of local runs. Recall that the symbolic representation has two main components:

(i) the *isomorphism type* of the artifact variables, describing symbolically the structure of the portion of the database reachable from the variables by navigating foreign keys

(ii) for each artifact relation and isomorphism type, the number of tuples in the relation that share that isomorphism type

Observe that because of (ii), the symbolic representation is not finite state. Indeed, (ii) requires maintaining a set of counters, which can grow unboundedly.

The heart of the proof in Chapter 3 is showing that it is sufficient to verify symbolic runs rather than actual runs. That is, for every LTL-FO formula φ_f, all local runs of T satisfy φ_f iff all symbolic local runs of T satisfy φ_f. Then the verification algorithm checks that there is no symbolic local run of T violating φ_f (so satisfying $\neg\varphi_f$). The algorithm relies on a reduction to (repeated) state reachability in VASS. This turns out to be sufficient to capture the information described above. The states of the VASS correspond to the isomorphism types of the artifact variables, combined with states of the Büchi automaton needed to check satisfaction of $\neg\varphi$.

The above approach can be viewed as symbolically running the HAS* specification. Consider the example in Section 5.2. After the **TakeOrder** task is called and returned, one possible local run of **ProcessOrders** might impose a set of constraints $\{\texttt{item\_id} \neq \texttt{null}, \texttt{cust\_id} \neq \texttt{null}, \texttt{status} = \text{“OrderPlaced”}, \texttt{instock} = \text{“Yes”}\}$ onto the artifact tuple of **ProcessOrders**. Now suppose the **CheckCredit** task is called. The local run can make the choice that the customer has good credit. Then when **CheckCredit** returns, the above set of constraints is updated with constraint $\{\texttt{cust\_id.record.status} = \text{"Good"}\}$, which means that in the read-only database, the credit record referenced by `cust_id` via foreign key satisfies $\texttt{status} = \text{"Good"}$. Next, suppose the *StoreOrder* service is applied in **ProcessOrders**. Then we symbolically store the current set of constraints by increasing its corresponding counter by 1. The set of constraints of the artifact tuple is reset to $\{\texttt{item\_id} = \texttt{null}, \texttt{cust\_id} = \texttt{null}, \texttt{status} = \text{"Init"}\}$ as specified in the post-condition of *StoreOrder*.

Although decidability of verification can be shown as outlined above, implementation of an efficient verifier is challenging. The algorithm that directly translates the artifact specification and the LTL-FO property into VASS's and checks (repeated) reachability is impractical because the resulting VASS can have exponentially many states and counters in the input size, and state-of-the-art VASS tools can only handle a small number of counters ($<$100) [3]. To mitigate the inefficiency, VERIFAS never generates the whole VASS but instead lazily computes the symbolic representations on-the-fly. Thus, it only generates *reachable* symbolic states, whose number is usually much smaller. In addition, isomorphism types in the symbolic representation are replaced by *partial isomorphism types*, which store only the subset of constraints on the variables imposed by the current run, leaving the rest unspecified. This representation is not only more compact, but also results in a significantly smaller search space in practice.

In the rest of the section, we first introduce our revised symbolic representation based on partial isomorphism types. Next, we review the classic Karp-Miller algorithm adapted to the symbolic version of HAS* for solving state reachability problems. Three specialized optimizations are introduced to improve the performance. In addition, we show that our algorithm with the optimizations can be extended to solve the repeated state reachability problems so that full LTL-FO verification of infinite runs can be carried out. For clarity, the exposition in this section focuses on specifications with a single task. The actual implementation extends these techniques to the full model with arbitrary number of tasks.

5.3.2 Partial Isomorphism Types

We start with our symbolic representation of local runs using partial isomorphism types. Intuitively, a partial isomorphism type captures the necessary constraints imposed by the current run on the current artifact tuple and the read-only database. Compared to the full isomorphism types in HAS or TAS, a partial isomorphism type allows "missing" edges between two expressions, meaning that it is unknown whether the two expressions are equal or not. This results in a more compact representation in practice, as shown by our experiments.

A partial isomorphism type contains a set of expressions that denote variables, constants and navigation via foreign keys from id variables or attributes. An expression is either:

- a constant c occurring in Γ or φ_f, or
- a sequence $\xi_1.\xi_2.\dots.\xi_m$, where ξ_1 is an id artifact variable x or an id attribute A of some artifact relation $\mathcal{S}$, ξ_2 is an attribute of $R \in \mathcal{DB}$ where $R.ID = \texttt{type}(\xi_1)$, and for each i, $2 \leq i < m$, ξ_i is a foreign key and ξ_{i+1} is an attribute in the relation referenced by ξ_i.

We denote by $\mathcal{E}$ the set of all expressions. Note that there is a subtle difference with the definition of expressions in HAS: the head ξ_1 of an expression can also be an attribute of an artifact relation. This is because each attribute of an artifact relation is no longer bound to a fixed variable as in HAS, so we also need to capture navigation starting from attributes.

Note that the length of expressions is bounded because of the acyclicity of the foreign keys, so $\mathcal{E}$ is finite.

We can now define partial isomorphism types.

Definition 127. *A* ***partial isomorphism type*** τ *is an undirected graph over* $\mathcal{E}$ *with each edge labeled by* $=$ *or* $\neq$*, such that the equivalence relation* $\sim$ *over* $\mathcal{E}$ *induced by the edges labeled with* $=$ *satisfies:*

1. *for every* $e, e' \in \mathcal{E}$ *and every attribute* A*, if* $e \sim e'$ *and* $\{e.A, e'.A\} \subseteq \mathcal{E}$ *then* $e.A \sim e'.A$*, and*
2. $(e_1, e_2, \neq) \in \tau$ *implies that* $e_1 \not\sim e_2$ *and for every* $e_1' \sim e_1$ *and* $e_2' \sim e_2$*,* $(e_1', e_2', \neq) \in \tau$*.*

Intuitively, a partial isomorphism type keeps track of a set of "$=$" and "$\neq$" constraints and their implications among $\mathcal{E}$. Condition 1 guarantees satisfaction of the key and foreign key dependencies. Condition 2 guarantees that there is no contradiction among the $\neq$-edges and the $=$-edges. In addition, the connection between two expressions can also be "unknown" if they are not connected by an edge. The full isomorphism type can be viewed as a special case of partial isomorphism type where the undirected graph is complete. In the worst case, the total number of partial isomorphism types is no smaller than the number of full isomorphism types so using partial isomorphism types does not improve the complexity upper bound. In practice, however, since the number of constraints imposed by a run is likely to be small, using partial isomorphism types can greatly reduce the search space.

Example 128. *Figure 5.4 shows two partial isomorphism types τ_1 (left) and τ_2 (right), where $R(\text{ID}, A)$ is the only database relation and $\{x, y, z\}$ are 3 variables of type $R.\text{ID}$. Solid lines are $=$-edges and dashed lines are $\neq$-edges. In τ_1, (x, y) is connected with $=$ so the edge $(x.A, y.A, =)$ is enforced by the key dependency. Missing edges between $(x.A, z.A)$ and $(y.A, z.A)$ indicate these connections are "unknown". τ_2 is a full isomorphism type, which requires the graph to be complete so $(x.A, z.A)$, $(y.A, z.A)$ and all pairs between $\{x, y, z\}$ and $\{x.A, y.A, z.A\}$ must be connected by either $=$ or $\neq$. The $\neq$-edges between $\{x, y, z\}$ and $\{x.A, y.A, z.A\}$ are omitted in the figure for clarity.*

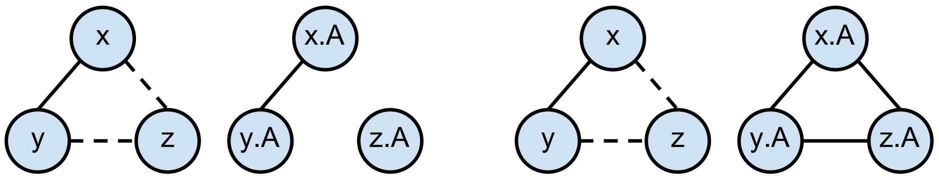


Figure 5.4. Partial and Full Isomorphism Types

We next define *partial symbolic instances*. Intuitively, a partial symbolic instance consists of a partial isomorphism type capturing the connections of the current tuple of $\bar{x}$, as well as, for the tuples present in the artifact relations, the represented isomorphism types t and the count of tuples sharing t.

Definition 129. *A **partial symbolic instance** I is a tuple $(\tau, \bar{c})$ where τ is a partial isomorphism type and $\bar{c}$ is a vector of $\mathbb{N}$ where each dimension of $\bar{c}$ corresponds to a unique partial isomorphism type.*

It turns out that most of the dimensions of $\bar{c}$ equal 0 in practice, so in implementation we only materialize a list of those dimensions with positive counter values. We denote by $\texttt{pos}(\bar{c})$ the set $\{\tau_S | \bar{c}(\tau_S) > 0\}$.

Next, we define *symbolic transitions* among partial symbolic instances by applying internal services. First we need to define condition evaluation on partial isomorphism types. Given a partial isomorphism type τ, satisfaction of a condition ϕ in negation normal form[1] by τ, denoted $\tau \models \phi$, is defined as follows:

- $x \circ y$ holds in τ iff $(x, y, \circ) \in \tau$ for $\circ \in \{=, \neq\}$,
- for relation $R(\textit{ID}, A_1, \ldots, A_m)$, $R(x, y_1, \ldots, y_m)$ holds in τ iff $(y_i, x.A_i, =) \in \tau$ for every $1 \leq i \leq m$,
- $\neg R(x, y_1, \ldots, y_m)$ holds in τ iff $(y_i, x.A_i, \neq) \in \tau$ for some $1 \leq i \leq m$, and
- Boolean combinations of conditions are standard.

Notice that $\tau \not\models \phi$ might be due to missing edges in τ but not because of inconsistent edges, so it is possible to satisfy ϕ by filling in the missing edges. This is captured by the notion of extension. We call τ' an *extension* of τ if $\tau \subseteq \tau'$ and τ' is consistent, meaning that the edges in τ' do not imply any contradiction of (in)equalities. We denote by $\texttt{eval}(\tau, \phi)$ the set of all *minimal extensions* τ' of τ such that $\tau' \models \phi$. Intuitively, $\texttt{eval}(\tau, \phi)$ contains partial isomorphism types obtained by augmenting τ with a minimal set of constraints to satisfy ϕ.

A symbolic transition is defined informally as follows (the full definition can be found in Appendix 5.6). To make a symbolic transition with a service $\sigma = (\pi, \psi, \bar{y}, \delta)$ from $I = (\tau, \bar{c})$ to $I' = (\tau', \bar{c}')$, we first extend the partial isomorphism type τ to a new partial isomorphism type τ_0 to satisfy the pre-condition π. Then the constraints on the propagated variables $\bar{y}$ are preserved by computing τ_1, the projection of τ_0 onto $\bar{y}$. Intuitively, the projection keeps only the

[1]Negations are pushed down to leaf atoms.

expressions headed by variables in $\bar{y}$ and their connections. Finally, τ' is obtained by extending τ_1 to satisfy the post-condition ψ. If δ is an insertion, then the counter that corresponds to the partial isomorphism type of the inserted tuple is incremented. If δ is a retrieval, then a partial isomorphism type τ_S with positive count is chosen nondeterministically and its count is decremented. The new partial isomorphism type τ' is then extended with the constraints from τ_S. We denote by $\texttt{succ}(I)$ the set of possible successors of I by taking one symbolic transition with any service σ.

Example 130. *Figure 5.5 shows an example of symbolic transition. The DB schema is that of Example 128. The variables are x, y of type $R.\mathrm{ID}$ and a non-ID variable z, with input variables $\{y, z\}$. The applied service is $\sigma = (\pi : R(x, z), \psi : x \neq y, \bar{y} : \{y, z\}, \delta : \{-\mathcal{S}(x, z)\})$. First, the pre-condition $R(x, z)$ is evaluated so edge $(x.A, z, =)$ is added (top-middle). Edge $(y.A, z, \neq)$ is also added so that the partial isomorphism type remains valid. Then variables $\{y, z\}$ are propagated, so the edges related to x or $x.A$ are removed (top-right). Next, we evaluate the post-condition $x \neq y$ so $(x, y, \neq)$ is added (bottom-right). Finally, a tuple from S is retrieved and overwrites $\{x, z\}$. Suppose the nondeterministically chosen τ_S contains a single edge $(x.A, z, =)$ (below the retrieve arrow). Then $\bar{c}(\tau_S)$ is decremented and τ_S is merged into the final partial isomorphism type (bottom left). Note that if σ contains an insertion of $+S(x, z)$ in δ instead of a retrieval, then the subgraph of τ_0 (top-middle) projected to $\{x, z\}$ is inserted to S. The corresponding counter in $\bar{c}$ will be incremented by 1.*

With symbolic transitions in place, verification works as follows. Informally, given a single-task HAS* Γ and a LTL-FO property φ_f, one can check whether $\Gamma \models \varphi_f$ by constructing a new HAS* Γ' obtained by combining Γ with conditions in φ_f and the Büchi automaton $B_{\neg\varphi}$ built from $\neg\varphi$. We can show that deciding whether $\Gamma \not\models \varphi_f$ reduces to checking whether an accepting state of $B_{\neg\varphi}$ is repeatedly reachable in Γ'. Verification therefore amounts to solving the following problem:

Problem 131. *(Symbolic Repeated Reachability, or SRR) Given a HAS* Γ, an initial partial*

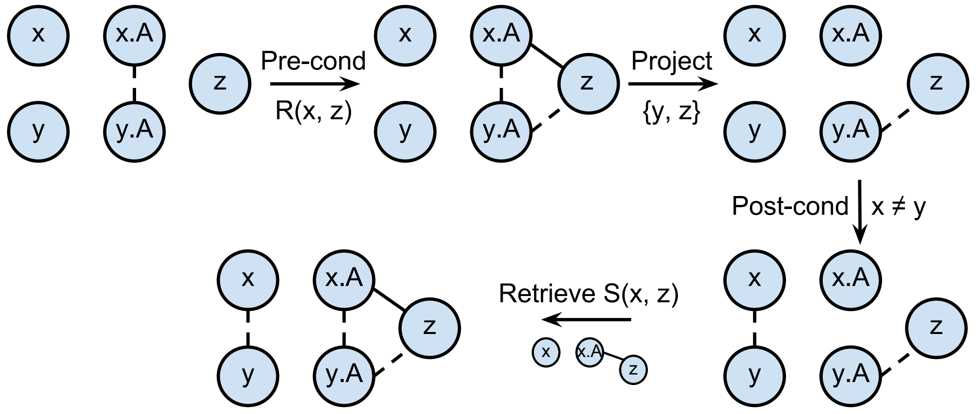


Figure 5.5. Symbolic Transition

symbolic state I_0, and a condition ϕ, is there a partial symbolic run $\{I_i\}_{0 \leq i \leq m < n}$ of Γ such that $I_{i+1} \in \texttt{succ}(I_i)$ for every $i \geq 0$, $\tau_m = \tau_n$, $\bar{c}_m \leq \bar{c}_n$ and $\tau_n \models \phi$?

The condition ϕ above simply states that $B_{\neg\varphi}$ is in one of its accepting states.

5.3.3 The Classic Karp-Miller Algorithm

The SRR Problem defines an infinite search space due to the unbounded counter values, so reduction to finite-state model checking is not possible. Adapting the theory developed in Chapter 3 from symbolic representation based on isomorphism types to symbolic representation based on partial isomorphism types, we can show that the symbolic transitions defined in Section 5.3.2 can be modeled as a VASS whose states are the partial symbolic instances of Definition 129. Consequently, The SRR problem reduces to testing (repeated) state reachability in this VASS. The benefit of the new approach is that this VASS is likely to have much fewer states and counters than the one defined in Chapter 3, because our search materializes partial isomorphism types parsimoniously, by lazily expanding the current partial type c using the (typically few) constraints of the symbolic transition to obtain a successor s. The transition from c to s concisely represents all the transitions from full-type expansions of c to full-type expansions of s (exponentially many in the number of “unknown” connections in c and s), which in the worst case would be individually explored by the algorithm in Chapter 3.

The VERIFAS implementation of the (repeated) state reachability is based on a series of optimizations to the classic Karp-Miller algorithm [83]. We describe the original Karp-Miller algorithm first, addressing the optimizations subsequently.

The Karp-Miller algorithm constructs a finite representation which over-approximates the entire (potentially infinite) reachable VASS state space, called a "coverability set" of states [61]. Any coverability set captures sufficient information about the global state space to support global reasoning tasks, including repeated reachability. In our context, the VASS states are the partial symbolic instances (PSIs) and a *coverability* set is a finite set $\mathcal{I}$ of PSIs, each reachable from the initial PSI I_0, such that for every reachable PSI $I = (\tau, \bar{c})$, there exists $I' = (\tau', \bar{c}') \in \mathcal{I}$ with $\tau = \tau'$ and $\bar{c} \leq \bar{c}'$. We say that I' *covers* I, denoted $I \leq I'$. To represent counters that can increase forever, the coverability set also allows an extension of PSIs in which some of the counters can equal ω. Recall that the ordinal ω is a special constant where $n < \omega$ for all $n \in \mathbb{N}$, $\omega \leq \omega$ and $\omega \pm 1 = \omega$.

Since the coverability set $\mathcal{I}$ is finite, we can effectively extract from it the reachable τ_n's that satisfy the condition ϕ (referring to the notation of the SRR problem). To test whether τ_n is repeatedly reachable, we can show that I_n is repeatedly reachable iff I_n is contained in a cycle consisting of only states in $\mathcal{I}$. As a result, the repeatedly reachable τ_n's can be found by constructing the transition graph among $\mathcal{I}$ and computing its *strongly connected components*. A partial isomorphism type τ is repeatedly reachable if its corresponding PSI I is included in a component containing a non-trivial cycle.

The Karp-Miller algorithm searches for a coverability set by materializing a finite part of the (potentially infinite) VASS transition graph starting from the initial state and executing transitions, pruning transitions to states that are covered by already materialized states. The resulting transition subgraph is traditionally called the *Karp-Miller tree* (despite the fact that it is actually a DAG).

In the notation of the SRR problem, note that if at least one counter value strictly increases from I_m to I_n ($\bar{c}_m^i < \bar{c}_n^i$ for some dimension i), then the sequence of transitions from m to n can repeat indefinitely, periodically reaching states with the same partial isomorphism type τ_n, but with ever-increasing associated counter values in dimension i (there are infinitely many such states). In the limit, the counter value becomes ω so a coverability set must include a state $(\tau_n, \bar{c})$ with $\bar{c}^i = \omega$, covering these infinitely many states.

Finite construction of the tree is possible due to a special *accelerate* operation that skips directly to a state with ω-valued counters, avoiding the construction of the infinitely many states it covers. Adapted to our context, when the algorithm detects a path $\{I_i\}_{0 \leq i \leq m < n}$ in the tree where $I_m \leq I_n$, the accelerate operation replaces in I_n the values of $\bar{c}_n(\tau_S)$ with ω for every τ_S where $\bar{c}_m(\tau_S) < \bar{c}_n(\tau_S)$.

We outline the details in Algorithm 1, which outputs the Karp-Miller tree $\mathcal{T}$. We denote by $\texttt{ancestors}(I)$ the set of ancestors of I in $\mathcal{T}$. Given a set $\mathcal{I}$ of states and a state $I' = (\tau', \bar{c}')$, the accelerate function is defined as $\texttt{accel}(\mathcal{I}, I') = (\tau', \bar{c}'')$ where for every τ_S, $\bar{c}''(\tau_S) = \omega$ if there exists $(\tau, \bar{c}) \in \mathcal{I}$ such that $\tau = \tau'$, $\bar{c} \leq \bar{c}'$ and $\bar{c}(\tau_S) < \bar{c}'(\tau_S)$. Otherwise, $\bar{c}''(\tau_S) = \bar{c}'(\tau_S)$.

Algorithm 1: Karp-Miller Tree Search Algorithm

```
input     : Initial instance $I_0$
output    : $\mathcal{T}$, the Karp-Miller tree
variables : $W$, set of states waiting to be explored
1 $W \leftarrow \{I_0\}, \mathcal{T} \leftarrow (\{I_0\}, \emptyset)$;
2 while $W \neq \emptyset$ do
3     Remove a state $I$ from $W$;
4     for $I' \in \texttt{succ}(I)$ do
5         $I'' \leftarrow \texttt{accel}(\texttt{ancestors}(I), I')$;
6         if $I'' \notin \mathcal{T} \vee I'' \in W$ then
7             Add edge $(I, I'')$ to $\mathcal{T}$;
8             $W \leftarrow W \cup \{I''\}$;
9 Return $\mathcal{T}$;
```

5.3.4 Optimization with Monotone Pruning

The original Karp-Miller algorithm is well-known to be inefficient in practice due to state explosion. To improve performance, various techniques have been proposed. The main technique we adopted in VERIFAS is based on pruning the Karp-Miller tree by monotonicity.

Intuitively, when a new state $I = (\tau, \bar{c})$ is generated during the search, if there exists a visited state I' where $I \leq I'$, then I can be discarded because for every state $\tilde{I}$ reachable from I, there exists a reachable state $\tilde{I}'$ starting from I' such that $\tilde{I} \leq \tilde{I}'$ by applying the same sequence of transitions that leads I to $\tilde{I}$. For the same reason, if $I \geq I'$, then I' and its descendants can be pruned from the tree. However, correctness of pruning is sensitive to the order of application of these rules (for example, as illustrated in [62], application of the rules in a breadth-first order may lead to incompleteness). The problem of how to apply the rules without losing completeness was studied in [62, 108] and we adopt the approach in [108]. More specifically, Algorithm 1 is extended by keeping track of a set act of "active" states and adding the following changes:

- Initialize act with $\{I_0\}$;
- In line 3, choose the state from $W \cap \texttt{act}$;
- In line 5, accel is applied on $\texttt{ancestors}(I) \cap \texttt{act}$;
- In line 8, I'' is not added to W if there exists $\hat{I} \in \texttt{act}$ such that $I'' \leq \hat{I}$;
- When I'' is added to W, remove from act every state $\hat{I}$ and its descendants for $\hat{I} \leq I''$ and $\hat{I}$ is either active or not an ancestor of I''. Add I'' to act.

5.3.5 A Novel, More Aggressive Pruning

We generalize the comparison relation $\leq$ of partial symbolic instances to achieve more aggressive pruning of the explored transitions. The novel comparison is based on the insight that a state I can be pruned in favor of I' as long as every partial isomorphism type reachable from I is also reachable from I'. So $I = (\tau, \bar{c})$ can be pruned by $I' = (\tau', \bar{c}')$ if τ' is "less restrictive" than τ (or τ implies τ'), and for every occurrence of τ_S in $\bar{c}$, there exists a corresponding occurrence of τ'_S in $\bar{c}'$ such that τ'_S is "less restrictive" than τ_S. Formally, given partial isomorphism types τ and τ', τ implies τ', denoted as $\tau \models \tau'$, iff $\tau' \subseteq \tau$. We replace the coverage relation $\leq$ on partial symbolic instances with a new binary relation $\preceq$ as follows.

Definition 132. *Given two partial symbolic states $I = (\tau, \bar{c})$ and $I' = (\tau', \bar{c}')$, $I \preceq I'$ iff $\tau \models \tau'$ and there exists $f : \texttt{pos}(\bar{c}) \times \texttt{pos}(\bar{c}') \mapsto \mathbb{N} \cup \{\omega\}$ such that*

- $f(\tau_S, \tau'_S) > 0$ *only when* $\tau_S \models \tau'_S$,
- *for every* τ_S, $\sum_{\tau'_S} f(\tau_S, \tau'_S) = \bar{c}(\tau_S)$ *and*
- *for every* τ'_S, $\sum_{\tau_S} f(\tau_S, \tau'_S) \leq \bar{c}'(\tau'_S)$.

Intuitively, f describes a one-to-one mapping from tuples stored in the artifact relations in I to tuples in I'. $f(\tau_S, \tau'_S) = k$ means that there are k tuples in I of partial isomorphism type τ_S that are mapped to k tuples in I' of type τ'_S. The condition $\tau_S \models \tau'_S$ guarantees that each tuple in I is mapped to one in I' of a less restrictive type.

Example 133. *Consider the two PSIs* $I = (\tau, \bar{c} = \{\tau_a : 2, \tau_b : 2\})$ *(left) and* $I' = (\tau', \bar{c}' = \{\tau_a : 3, \tau_b : 1\})$ *(right) shown in Figure 5.6. Since* $\tau \neq \tau'$ *and* $\bar{c}(\tau_b) > \bar{c}'(\tau_b)$, $I \leq I'$ *does not hold. However, any sequence of symbolic transitions applicable starting from* I *can also be applied starting from* I', *because if the conditions imposed by these transitions do not conflict with those in* I, *then they won't conflict with the subset thereof in* I'. *Consequently,* I *can be pruned if* I' *is found during the search. This fact is detected by* $\preceq$: $I \preceq I'$ *holds since* $\tau \models \tau'$ *and we can construct* f *as* $f(\tau_a, \tau_a) = 2$ *and* $f(\tau_b, \tau_b) = f(\tau_b, \tau_a) = 1$ *since* $\tau_b \models \tau_a$.

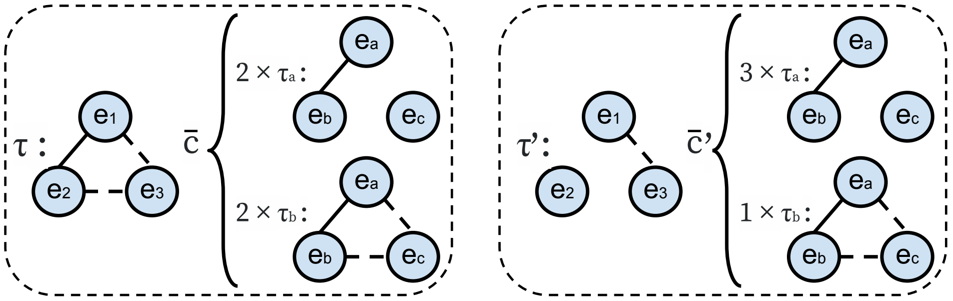


Figure 5.6. Illustration of $\preceq$

Note that one can efficiently test whether $I \preceq I'$ by reduction to the Max-Flow problem over a flow graph F with node set $\texttt{pos}(\bar{c}) \cup \texttt{pos}(\bar{c}') \cup \{s, t\}$, with s a source node and t a sink node. For every $\tau_S \in \texttt{pos}(\bar{c})$, there is an edge from s to τ_S with capacity $\bar{c}(\tau_S)$. For every $\tau'_S \in \texttt{pos}(\bar{c}')$, there is an edge from τ'_S to t with capacity $\bar{c}'(\tau_S)$. For every pair of τ_S, τ'_S, there is an edge from τ_S to τ'_S with capacity ∞ if $\tau'_S \models \tau_S$. We can show that F has a max-flow equal to $\sum_{\tau_S} \bar{c}(\tau_S)$ if and only if $I \preceq I'$.

The same idea can also be applied to the `accel` function. Formally, given $\mathcal{I}$ and $I' = (\tau', \bar{c}')$, the new accelerate function
$\texttt{accel}(\mathcal{I}, I') = (\tau', \bar{c}'')$ where $\bar{c}''(\tau'_S) = \omega$ if there exists $I \in \mathcal{I}$ such that $I \preceq I'$ and there exists mapping f satisfying the conditions in Definition 132 and $\sum_{\tau_S} f(\tau_S, \tau'_S) < \bar{c}'(\tau'_S)$. Otherwise $\bar{c}''(\tau'_S) = \bar{c}'(\tau'_S)$.

5.3.6 Data Structure Support

The above optimization relies on two important operations applied every time a new state is explored: given the set of active states `act` and a partial symbolic state I, (1) compute the set $\{I' | I' \preceq I \wedge I' \in \texttt{act}\}$ and (2) check whether there exists $I' \in \texttt{act}$ such that $I \preceq I'$. As each test of $\preceq$ might require an expensive operation of computing the max-flow, when $|\texttt{act}|$ is large, checking whether $I' \preceq I$ (or $I \preceq I'$) for every $I' \in \texttt{act}$ would be too time-consuming.

We start with the simple case where $\bar{c} = \bar{0}$ in all I's. Then to test whether $I \preceq I'$ for $I = (\tau, \bar{c})$ and $I' = (\tau', \bar{c}')$ is to test whether $\tau' \subseteq \tau$. When the partial isomorphism types are stored as sets of edges, we can accelerate the two operations with data structures that support fast subset (superset) queries: given a collection $\mathcal{C}$ of sets and a query set q, find all sets in $\mathcal{C}$ that are subsets (supersets) of q. The standard solutions are to use Tries for superset queries [109] and Inverted Lists for subset queries [93].

The same idea can be applied to the general case where $\bar{c} \geq 0$ to obtain an over-approximations of the precise results. Given $I = (\tau, \bar{c})$, we let $E(I)$ be the set of edges in τ or any τ_S where $\bar{c}(\tau_S) > 0$. Then given I and I', $I \preceq I'$ implies $E(I') \subseteq E(I)$. We build the Trie and Inverted Lists indices such that for a given query I, they return a candidate set $\mathcal{I}^{\preceq}$ and a candidate set $\mathcal{I}^{\succeq}$ where $\mathcal{I}^{\preceq}$ contains all I' from `act` such that $E(I') \subseteq E(I)$ and $\mathcal{I}^{\succeq}$ contains all I' such that $E(I) \subseteq E(I')$. Then it suffices to test each member in the candidate sets for $I \preceq I'$ and $I \succeq I'$ to obtain the precise results of operations (1) and (2).

5.3.7 Optimization with Static Analysis

Next, we introduce our optimization based on static analysis. At a high level, we notice that in real workflow examples, some constraints in conditions of the specification and the property are irrelevant to the result of verification because they can never cause violations when conditions are evaluated in a symbolic run. Such conditions can be ignored to reduce the number of symbolic states. For example, for a constraint $x = y$ in the specification, if $x \neq y$ does not appear anywhere else and cannot be implied by other constraints, then $x = y$ can be safely removed from any partial isomorphism types without affecting the result of the verification algorithm. Our goal is to detect all such constraints by statically analyzing the HAS* and the LTL-FO property. Specifically, we analyze the constraint graph consisting of all possible "=" and "$\neq$" constraints that can potentially be added to any partial isomorphism types in symbolic transitions of the HAS* Γ or when checking condition ϕ (refer to the notation in the SRR problem).

Definition 134. *The constraint graph G of (Γ, ϕ) is a labeled undirected graph over the set of all expressions $\mathcal{E}$ with the following edges. For every atom a that appears in a condition of Γ or ϕ in negation normal form, if a is*

- $(x = y)$*, then G contains $(x.w, y.w, =)$ for all sequences w where $\{x.w, y.w\} \subseteq \mathcal{E}$,*
- $(x \neq y)$*, then G contains $(x, y, \neq)$,*
- $R(x, y_1, \ldots, y_m)$*, then G contains $(x.A_i.w, y_i.w, =)$ for all i and sequences w where $\{x.A_i.w, y_i.w\} \subseteq \mathcal{E}$, and*
- $\neg R(x, y_1, \ldots, y_m)$*, then G contains $(x.A_i, y_i, \neq)$ for all i.*

For any subgraph G' of G, G' is consistent *if the edges in G' do not imply any contradiction, meaning that there is no path of $=$-edges connecting two distinct constants or two expressions connected by an $\neq$-edge.*

An edge e of G is non-violating *if for every consistent subgraph G', $G' \cup \{e\}$ is also consistent.*

Intuitively, by collecting the edges described above, the constraint graph G becomes an over-approximation of the reachable partial isomorphism types. Thus any edge in G that is non-violating is also non-violating in any reachable partial isomorphism type. So our goal is to find all the non-violating edges in G, since they can be ignored in partial isomorphism types to reduce the size of the search space.

Non-violating edges can be identified efficiently in polynomial time. Specifically, an edge $(u, v, \neq)$ is non-violating if u and v belong to different connected components of $=$-edges of G. An $=$-edge e is non-violating if there is no path $u \longrightarrow v$ of $=$-edges containing e for any $(u, v, \neq) \in G$ or (u, v) being two distinct constants. This can be checked efficiently by computing the biconnected components of the $=$-edges [121]. We omit the details here.

Example 135. *Consider the two constraint graphs G_1 and G_2 in Figure 5.7. In G_1 (left), (e_3, e_5) is a non-violating $\neq$-edge because e_3 and e_5 belong to two different connected components of $=$-edges ($\{e_1, e_2, e_3, e_4\}$ and $\{e_5, e_6, e_7\}$ respectively). In G_2 (right), (e_3, e_5) is a non-violating $=$-edge because (e_3, e_5) is not on any simple path of $=$-edges connecting the two ends of any $\neq$-edges (i.e. (e_2, e_3) and (e_5, e_6)).*

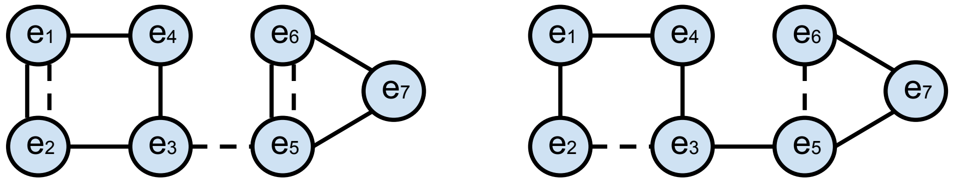


Figure 5.7. Non-violating Edges

5.3.8 Extension to Repeated-Reachability

Recall from Section 5.3.2 that providing full support for verifying LTL-FO properties requires solving the repeated state reachability problem. It is well-known that for VASS, the coverability set $\mathcal{I}$ extracted from the tree $\mathcal{T}$ constructed by the classic Karp-Miller algorithm can be used to identify the repeatedly reachable partial isomorphism types (see Section 5.3.3). The same idea can be extended to the Karp-Miller algorithm with monotone pruning (Section 5.3.4), since the algorithm is guaranteed to construct a coverability set.

However, the Karp-Miller algorithm equipped with our $\preceq$-based pruning (Section 5.3.5) explores fewer states due to the more aggressive pruning, and it turns out that the resulting coverability set $\mathcal{I}^{\preceq}$ is incomplete to determine whether a state is repeatedly reachable. We can no longer guarantee that a repeatedly reachable state is only contained in a cycle of states in $\mathcal{I}^{\preceq}$. We can nevertheless show that the completeness of the search for repeatedly reachable states can be restored by developing our own extraction technique which compensates for the overly aggressive $\preceq$-based pruning. The technical development is subtle and relegated to Appendix 5.7. As confirmed by our experimental results, the additional overhead is acceptable.

5.4 Experimental Evaluation

We evaluated the performance of VERIFAS using both real-world and synthetic artifact specifications.

5.4.1 Setup and Benchmark

The Real Set. We built an artifact system benchmark by rewriting in HAS* a sample of real-world BPMN workflows published at the official BPMN website [1], which provides 36 workflows of non-trivial size. To rewrite these workflows in HAS*, we manually added the database schema, artifact variables/relations, and services for updating the data. HAS* is sufficiently expressive to specify 32 of the 36 BPMN workflows. The remaining 4 cannot be expressed in HAS* because they involve computing aggregate functions or updating unboundedly many tuples of the artifact relations, which is not supported in the current model. We will consider such features in our future work.

The Synthetic Set. Since we wished to stress-test VERIFAS, we also randomly generated a set of HAS* specifications of increasing complexity. All components of each specification, including DB schema, task hierarchy, variables, services and conditions, were generated fully at random for a certain size. We provide in Appendix 5.8 more details on how each specification is generated. Those with empty state space due to unsatisfiable conditions were removed from the benchmark. Table 5.1 shows some statistics of the two sets of specifications. (#Relations, #Tasks, etc. are averages over the real / synthetic sets of workflows.)

Table 5.1. Statistics of the Two Sets of Workflows

Dataset	Size	#Relations	#Tasks	#Variables	#Services
Real	32	3.563	3.219	20.63	11.59
Synthetic	120	5	5	75	75

LTL-FO Properties. On each workflow of both sets, we run our verifier on a collection of 12 LTL-FO properties of the root task constructed using templates of real propositional LTL properties. The set of templates used here are the same classes used for SpinArt. The LTL properties are all the 11 examples of safety, liveness and fairness properties collected from a standard reference paper [116] and an additional property `False` used as a baseline when comparing the performance of VERIFAS on different classes of LTL-FO properties. We list all the templates of LTL properties in Table 5.4. To see why we choose `False` as the baseline property, recall from Section 5.3 that the verifier's running time is mainly determined by the size of the reachable symbolic state space (VERIFAS first computes all reachable symbolic states –represented by the coverability set– then identifies the repeatedly-reachable ones). The reachable symbolic state space can be conceptualized as the cross-product between the reachable symbolic state space of the HAS* specification (absent any property) and the Büchi automaton of the property. When the LTL-FO property is `False`, the generated Büchi automaton is of the simplest form (a single accepting state within a loop), so it has no impact on the cross-product size, unlike more complex properties.

In each workflow, we generate an LTL-FO property for each template by replacing the propositions with FO conditions chosen from the pre-and-post conditions and their sub-formulas. Note that by doing so, the generated LTL-FO properties on the real workflows are combinations of real propositional LTL properties and real FO conditions, and so are close to real-world LTL-FO properties.

Baseline. We compare VERIFAS with SpinArt, the verifier implementation based on Spin presented in Chapter 4. Note that SpinArt supports a restricted version of the HAS* model since it cannot handle the updatable artifact relations.

Platform. We implemented both verifiers in `C++` with Spin version 6.4.6 for SpinArt. All experiments were performed on a Linux server with a quad-core Intel i7-2600 CPU and 16G memory. For each specification, we ran our verifiers to test each of the 12 generated LTL-FO properties, resulting in 384 runs for the real set and 1440 runs for the synthetic set. Towards fair comparison, since SpinArt cannot handle artifact relations, in addition to running our full verifier (VERIFAS), we also ran it with artifact relations ignored (VERIFAS-NoSet). The timeout limit of each run was set to 10 minutes and the memory limit was set to 8G.

5.4.2 Experimental Results

Performance. Table 5.2 shows the results on both sets of workflows. SpinArt achieves acceptable performance in the real set, with an average elapsed time of a few seconds and only 3 timeouts. However, it failed in a large number of runs (440/1440) in the stress-test using synthetic specifications. On the other hand, both VERIFAS and VERIFAS-NoSet achieve average running times within 0.3 second and no timeout on the real set, and the average running time is within seconds on the synthetic set, with only 19 timeouts over 1440 runs. The presence of artifact relations introduced an acceptable amount of performance overhead, which was negligible in the real set and less than 60% in the synthetic set. Compared with SpinArt, VERIFAS is $>$10x faster in average running time and scales significantly better with workflow complexity.

The timeout runs on the synthetic workflows are all due to state explosion with a state space of size $\sim 3 \times 10^4$. The reason is that though unlikely in practice, it is still possible that the reached partial isomorphism types can degenerate to full isomorphism types, and in this case our state-pruning optimization does not reduce the number of reached states.

Table 5.2. Average Elapsed Time and Number of Failed Runs (#Fail) due to Timeout or Memory Overflow

Verifier	Real		Synthetic	
	Avg(Time)	#Fail	Avg(Time)	#Fail
SpinArt	2.97s	3	83.983s	440
VERIFAS-NoSet	**.229s**	0	**6.983s**	19
VERIFAS	**.245s**	0	**11.01s**	16

Cyclomatic Complexity. To better understand the scalability of VERIFAS, we also measured verification time as a function of workflow complexity, adopting a metric called *cyclomatic complexity*, which is widely used in measuring complexity of program modules [128]. For a program P with control-flow graph $G(V, E)$, the cyclomatic complexity of P equals $|E| - |V| + 2$. We adapt this measure to HAS* specifications as follows. Given a HAS* specification $\mathcal{A}$, a control flow graph of $\mathcal{A}$ can be obtained by selecting a task T of $\mathcal{A}$ and a non-id variable $x \in \bar{x}^T$ and projecting all services of T onto $\{x\}$. The resulting services contain only x and constants and thus can be viewed as a transition graph with x as the state variable. The cyclomatic complexity of $\mathcal{A}$, denoted as $M(\mathcal{A})$, is defined as the maximum cyclomatic complexity over all the possible control-flow graphs of $\mathcal{A}$ (corresponding to all possible projections).

Figure 5.8 shows that the verification time increases exponentially with the cyclomatic complexity, thus confirming the pertinence of the measure to predicting verification complexity, where the verification time of a workflow is measured by the average running time over all the runs of its LTL-FO properties. According to [128]'s recommendation, for a program to remain readable and testable, its cyclomatic complexity should not exceed 15. Among all the 138 workflows with cyclomatic complexity at most 15, VERIFAS successfully verified 130/138 ($\sim$94%) of them within 10s and only 4 instances have timeout runs (marked as hollow triangles in Figure 5.8). For specifications with complexity above 15, only 2/14 instances have timeout runs.

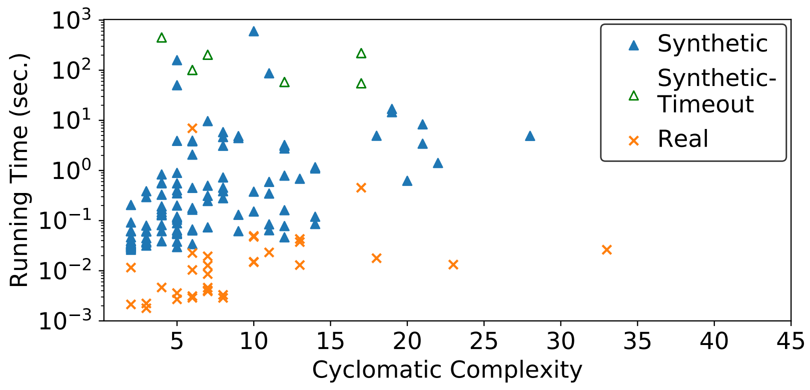


Figure 5.8. Average Running Time vs. Cyclomatic Complexity

Typically, for the same cyclomatic complexity, the real workflows can be verified faster compared to the synthetic workflows. This is because the search space of the synthetic workflows is likely to be larger because there are more variables and transitions.

Impact of Optimizations. We studied the effect of our optimization techniques: state pruning (SP, Section 5.3.5), data structure support (DSS, Section 5.3.6), and static analysis (SA, Section 5.3.7). For each technique, we reran the experiment with the optimization turned off and measured the speedup by comparing the elapsed verification time with the original elapsed time. Table 5.3 shows the average speedups of each optimization on both datasets. Since some instances have extreme speedups (over 10,000x), simply averaging could be misleading, so we also present the trimmed averages of the speedups (i.e. removing the top/bottom 5% speedups before averaging) to exclude the extreme values.

Table 5.3 shows that the effect of state pruning is the most significant in both sets of workflows, with an average (trimmed) speedup of ∼25x and ∼127x in the real and synthetic set respectively. The static analysis optimization is more effective in the real set (1.4x improvement) but its effect in the synthetic set is less pronounced. It creates a small amount (7%) of overhead in most cases, but significantly improves the running time of a single instance, resulting in the huge gap between the normal average speedup and the trimmed average speedup. The explanation to this phenomenon is that the workflows in the real set are more "sparse" in general, which means there are fewer comparisons within a subset of variables so a larger number of comparisons can be pruned by static analysis. Finally, the data-structure support provides $\sim$1.2x and $\sim$1.6x average speedup in each set respectively. Not surprisingly, the optimization becomes more effective as the size of the state space increases.

Table 5.3. Mean and Trimmed Mean (5%) of Speedups

Dataset	SP		SA		DSS	
	Mean	Trim.	Mean	Trim.	Mean	Trim.
Real	1586.54x	24.69x	1.80x	1.41x	1.87x	1.24x
Synthetic	322.03x	127.35x	28.78x	0.93x	2.72x	1.58x

Overhead of Repeated-Reachability. We evaluated the overhead of computing the set of repeatedly-reachable states from the coverability set (Section 5.3.8) by repeating the experiment with the repeated reachability module turned off. Compared with the turned off version, the full verifier has an average overhead of **19.03%** on the real set and **13.55%** overhead on the synthetic set (overheads are computed over the non-timed-out runs).

Effect of Different Classes of LTL-FO Properties. Finally, we evaluate how the structure of LTL-FO properties affects the performance of VERIFAS. Table 5.4 lists all the LTL templates used in generating the LTL-FO properties and their intuitive meaning stated in [116]. For each template and for each set of workflows, we measure the average running time over all the runs with LTL-FO properties generated using the template. Table 5.4 shows that for each class of properties, the average running time is within 2x of the average running time for the simplest non-trivial property `False`. This is much better than the theoretical upper bound, which is linear in size of the Büchi automaton of the LTL formula. Some properties even have a shorter running time because, although the space of partial symbolic instances is enlarged by the Büchi automaton, more of the states may become unreachable due to the additional constrains imposed by the LTL-FO property.

Table 5.4. Average Running Time of Verifying Different Classes of LTL-FO Properties

Templates for LTL-FO	Meaning	Real	Synthetic
`False`	Baseline	0.26s	10.13s
$\mathbf{G}\varphi$	Safety	0.28s	10.26s
$(\neg\varphi\ \mathbf{U}\ \psi)$	Safety	0.28s	16.13s
$(\neg\varphi\mathbf{U}\psi) \land \mathbf{G}(\varphi \to \mathbf{X}(\neg\varphi\mathbf{U}\psi))$	Safety	0.30s	10.79s
$\mathbf{G}(\varphi \to (\psi \lor \mathbf{X}\psi \lor \mathbf{XX}\psi))$	Safety	0.29s	12.07s
$\mathbf{G}(\varphi \lor \mathbf{G}(\neg\varphi))$	Safety	0.30s	12.17s
$\mathbf{G}(\varphi \to \mathbf{F}\psi)$	Liveness	0.29s	16.81s
$\mathbf{F}\varphi$	Liveness	0.02s	6.44s
$\mathbf{GF}\varphi \to \mathbf{GF}\psi$	Fairness	0.30s	14.09s
$\mathbf{GF}\varphi$	Fairness	0.28s	6.91s
$\mathbf{G}(\varphi \lor \mathbf{G}\psi)$	Fairness	0.05s	9.64s
$\mathbf{FG}\varphi \to \mathbf{GF}\psi$	Fairness	0.28s	6.75s

5.5 Conclusion

We presented the implementation of VERIFAS, an efficient verifier of temporal properties for data-driven workflows specified in HAS*, a variant of the Hierarchical Artifact System model studied theoretically in Chapter 3. HAS* is inspired by the Business Artifacts framework introduced by IBM [76] and incorporated in OMG's CMMN standard [94, 17].

While the verification problem is EXPSPACE-complete, our experiments show that the theoretical worst case is unlikely in practice and that verification is eminently feasible. Indeed, VERIFAS achieves excellent performance (verification within seconds) on a practically relevant class of real-world and synthetic workflows (those with cyclomatic complexity in the range recommended by good software engineering practice), and a set of representative properties. The good performance of VERIFAS is due to an adaptation of our symbolic verification techniques developed in Chapter 3, coupled with the classic Karp-Miller algorithm accelerated with an array of nontrivial novel optimizations.

The experiments highlight the impact of the various optimizations and parameters of both the artifact systems and properties. The performance of VERIFAS significantly improves over SpinArt. On the real set of workflows, even when the tasks have updatable artifact relations, VERIFAS achieves an $>$10x improvement in the average running time (0.245 seconds) with no failed runs. Also on the synthetic set of workflows, it is able to scale with an average running time of 11.01 second and fails due to timeout on $<1\%$ of the runs. In addition, the experiments show that the verification time of VERIFAS increases exponentially with the input workflow's cyclomatic complexity, a classic metric for measuring the complexity of program modules. The performance is within a reasonable range when the cyclomatic complexity is below the maximal value recommended by software engineering practice. Thus, VERIFAS performs very well on practically relevant classes of artifact systems. Compared to the Spin-based verifier, it not only applies to a much broader class of artifacts but also has a decisive performance advantage even on the simple artifacts the Spin-based verifier is able to handle. To the best of our knowledge, VERIFAS is the first implementation of practical significance of an artifact verifier with full support for unbounded data.

5.6 Appendix: Symbolic transitions

In this section, we provide the formal definition of symbolic transitions. Although our discussion in Section 5.3 focuses primarily on single tasks, the notion of partial symbolic instances and symbolic transitions can be naturally extended to the full HAS* model. For clarity, we first define symbolic transitions for single tasks (ignoring interactions with children) and then extend the definition to multiple tasks (taking into account such interactions).

Single Tasks. For a single task, we define $\texttt{succ}(\tau, \bar{c})$ as follows. For a given quantifier-free FO formula φ, we denote by $\texttt{flat}(\varphi)$ the formula obtained by replacing each relational atom $R(x, y_1, \ldots, y_m)$ with $\bigwedge_{i=1}^m x.A_i = y_i$, and denote by $\texttt{conj}(\varphi)$ the set of conjuncts of the disjunctive normal forms of $\texttt{flat}(\varphi)$. Note that each literal in every $\texttt{conj}(\varphi)$ is positive as negations can be removed by inverting each $=$ and $\neq$. We let $t(\theta)$ to be the partial isomorphism type induced by θ for $\theta \in \texttt{conj}(\varphi)$. Given a PSI $I = (\tau, \bar{c})$, for every service $\sigma = (\pi, \psi, \bar{y}, \delta)$, for every $\theta_1 \in \texttt{conj}(\pi)$ where $\tau \cap t(\theta_1)$ is satisfiable (the pre-condition is satisfied), for every $\theta_2 \in \texttt{conj}(\psi)$, $\texttt{succ}(I)$ contains the following PSIs that are valid:

- $((\tau \cap t(\theta_1)|\bar{y}) \cap t(\theta_2), \bar{c})$ if $\delta = \emptyset$,
- $((\tau \cap t(\theta_1)|\bar{y}) \cap t(\theta_2), \bar{c}[\tau_S \to \bar{c}(\tau_S) + 1])$ for $\tau_S = f_{\bar{z} \to \mathcal{S}}(\tau|\bar{z})$ if $\delta = \{+S(\bar{z})\}$, or
- $((\tau \cap t(\theta_1)|\bar{y}) \cap t(\theta_2) \cap f_{\mathcal{S} \to \bar{z}}(\tau_S), \bar{c}[\tau_S \to \bar{c}(\tau_S) - 1])$ if $\delta = \{-S(\bar{z})\})$ for some τ_S where $\bar{c}(\tau_S) > 0$.

Extension to Multiple Tasks. For each task T, the definition of partial isomorphism type of T is the same as Definition 127, except the set $\mathcal{E}$ is replaced with $\mathcal{E}^T$, the set of all expressions of task T. Partial symbolic instances are defined as follows.

Definition 136. *A partial symbolic instance (PSI) I of a task T is a tuple $(\tau, \bar{c}, \bar{r})$ where*

- *τ is a partial isomorphism type of T,*
- *$\bar{c}$ is a vector of $\mathbb{N}$ where each dimension of $\bar{c}$ corresponds to a unique partial isomorphism type of T, and*
- *$\bar{r}$ is a mapping from $child(T)$ to the set $\{\texttt{active}, \texttt{inactive}\}$.*

Intuitively, the extra component $\bar{r}$ records the status and return information of child tasks of T. For a child task $T_c \in child(T)$, $\bar{r}(T_c)$ indicates whether T_c is active or not.

Symbolic transitions are specified by the successor function
$\texttt{succ}^T(I)$ for each task T, which is formally defined as follows. We first consider successors under internal services, then under opening and closing services of the children of T.

Internal Services. Given a PSI $I = (\tau, \bar{c}, \bar{r})$ of task T the definition of symbolic transitions with internal services resembles the definition in the single task case, with the extra conditions that $\bar{r}(T_c) = \texttt{inactive}$ for every child task $T_c \in child(T)$ and for every resulting PSI $(\tau', \bar{c}', \bar{r}')$, $\bar{r}' = \bar{r}$.

We next consider opening and closing services of T's children. For PSI $I = (\tau, \bar{c}, \bar{r})$ of task T, $\texttt{succ}^T(I)$ contains the following:

Opening service $\sigma^o_{T_c}$ with pre-condition π: (applicable if $\bar{r}(T_c) = \texttt{inactive}$)

$$\texttt{succ}^T(I) \text{ contains } (\tau \cap t(\theta), \bar{c}, \bar{r}[T_c \mapsto \texttt{active}])$$

for every $\theta \in \texttt{conj}(\pi)$ such that $\tau \cap \theta$ is satisfiable.

Closing service $\sigma^c_{T_c}$: (applicable if $\bar{r}(T_c) = \texttt{active}$)

$$\texttt{succ}^T(I) \text{ contains } (\tau \cap \tau_c, \bar{c}, \bar{r}[T_c \mapsto \texttt{inactive}])$$

for every partial isomorphism type τ_c of $\bar{x}^T_{T_c\uparrow}$ such that $\tau \cap \tau_c$ is satisfiable.

5.7 Appendix: Repeated-Reachability

We describe in more detail how our Karp-Miller based algorithm with pruning can be extended to state repeated reachability, thus providing full support for verifying LTL-FO properties. For VASS, it is well-known that the coverability set $\mathcal{I}$ extracted from the tree $\mathcal{T}$ constructed by the classic Karp-Miller algorithm can be used to identify the repeatedly reachable partial isomorphism types (see Section 5.3.3). The same idea can be extended to the Karp-Miller algorithm with monotone pruning (Section 5.3.4), since the algorithm guarantees to construct a coverability set. This can be done as follows.

First, for every $I = (\tau, \bar{c}) \in \mathcal{I}$, if there exists τ_S such that $\bar{c}(\tau_S) = \omega$, then I is inherently repeatedly reachable because the accelerate operation was applied to generate the ω. In addition, it is sufficient to consider only the *maximal* PSIs in $\mathcal{I}$. A PSI I is maximal if there is no state $I' \neq I$ in $\mathcal{I}$ and $I \preceq I'$. We denote by $\mathcal{I}_{\texttt{max}}$ the subset of maximal PSIs in $\mathcal{I}$ that contains no ω.

Then for PSI $I \in \mathcal{I}_{\texttt{max}}$, it is not difficult to show that I is repeatedly reachable iff I is contained in a cycle consisting of *only* PSIs in $\mathcal{I}_{\texttt{max}}$. As a result, the set of repeatedly reachable states can be computed by constructing the transition graph among $\mathcal{I}_{\texttt{max}}$ and computing its *strongly connected components*. A PSI I is repeatedly reachable if I is contained in a component with at least 1 edge.

The reason this works is the following. Suppose $I \in \mathcal{I}_{\mathtt{max}}$ and I is contained in a cycle $\{I_i\}_{a \le i \le b}$ where I_a is not maximal. Then there exists a reachable PSI I'_a where $I_a < I'_a$. So the sequence of transitions that leads I_a to I_b is also a valid sequence that can be applied starting from I'_a and results in a cycle $\{I'_i\}_{a \le i \le b}$. The transitions leave the counters unchanged, so $I'_i > I_i$ for every $i \in [a, b]$, which contradicts the assumption that I is maximal.

However, the same approach cannot be directly applied when the Karp-Miller algorithm is equipped with our $\preceq$-based pruning (Section 5.3.5). The algorithm explores fewer states due to the more aggressive pruning, and it turns out that the resulting coverability set $\mathcal{I}^{\preceq}$ is incomplete to determine whether a state is repeatedly reachable. We can no longer guarantee that a repeatedly reachable state is only contained in a cycle of maximal states in $\mathcal{I}^{\preceq}$. The above reasoning fails because it relies on the *strict monotonicity* property: for every PSI I and I' where $I < I'$, for every $I_{\mathtt{next}} \in \mathtt{succ}(I)$, there exists $I'_{\mathtt{next}} \in \mathtt{succ}(I')$ such that $I_{\mathtt{next}} < I'_{\mathtt{next}}$. This no longer holds when $<$ is replaced with $\prec$. Consequently, we need to explore PSIs outside of $\mathcal{I}_{\mathtt{max}}$ when we detect cycles. To avoid state explosion when the extra PSIs are explored, we make use of a pruning criterion obtained by slightly restricting $\preceq$ as follows.

Definition 137. *Given two partial symbolic states $I = (\tau, \bar{c})$ and $I' = (\tau', \bar{c}')$, $I \preceq^{+} I'$ iff $I = I'$ or the following hold:*

- $\tau \models \tau'$,
- *there exists f that satisfies the conditions of Definition 132, and*
- *there exists τ'_S such that $\sum_{\tau_S} f(\tau_S, \tau'_S) < \bar{c}'(\tau'_S)$.*

By using the same Karp-Miller-based algorithm with $\preceq^{+}$ in monotone pruning, we compute a set $\mathcal{I}^{+}_{\mathtt{max}}$ that satisfies the following: for every $I \in \mathcal{I}_{\mathtt{max}}$ and I' reachable from I, there exists $I'' \in \mathcal{I}^{+}_{\mathtt{max}}$ such that $I' \preceq^{+} I''$. Since $\preceq^{+}$ satisfies the strict monotonicity property, a PSI $I \in \mathcal{I}_{\mathtt{max}}$ is repeatedly reachable iff I is contained in a cycle that contains only PSIs in $\mathcal{I}^{+}_{\mathtt{max}}$, which can be checked efficiently by computing the strongly connected components. Note that when we compute $\mathcal{I}^{+}_{\mathtt{max}}$, we can accelerate the search by pruning a new PSI I if $I \preceq^{+} I'$ for any I' in the previously computed Karp-Miller tree. In addition, there is no need to apply the accelerate operator since it is sufficient to consider only PSIs with no ω. As confirmed by our experiments, the overhead for computing the set of repeatedly reachable PSIs is acceptable.

5.8 Appendix: Synthetic Workflow Generator

We briefly describe how the synthetic workflows used in our experiments were produced. Each part of a synthetic workflow is generated at random, for the given parameters #relations, #tasks, #variables and #services.

We first generate a random tree of fixed size as the acyclic database schema where each relation has a fixed number of (4) non-ID attributes. Then we generate a random tree of fixed size as the task hierarchy. Within each task, for each variable type (non-ID or ID of some relation), we uniformly generate the same number of variables. We randomly choose 1/10 of the variables as input variables and another 1/10 as output variables. Then we generate a fixed number of internal services for each task with randomly generated pre-and-post conditions (described next). With probability 1/3, the internal service has $\bar{y}$ propagating a randomly chosen subset (1/10) of the task's variables, or inserting a fixed tuple of variables into the artifact relation, or retrieving a tuple from the artifact relation.

Each condition of each service is also generated as a random tree. We first generate a fixed number of (5) atoms, where each atom has 1/3 of probability of having the form $x = y$, $x = c$ or $R(\bar{x})$, where $x, y, \bar{x}$ are variables chosen uniformly at random and c is a random constant from a fixed set. Each leaf is negated with probability 1/2. Then we generate the condition as a random binary tree with the atoms as the leafs of the tree. Each internal node of the binary tree is an $\wedge$-connective with probability 4/5 and an $\vee$-connective with probability 1/5. We chose to generate $\wedge$ with higher probability based on our observations of the real workflows.

Acknowledgement

This chapter contains material from “VERIFAS: A Practical Verifier for Artifact Systems” by Yuliang Li, Alin Deutsch and Victor Vianu in Proceedings of the VLDB Endowment 11,3 (November 2017) [89]. The dissertation author was the primary investigator of this paper.

Chapter 6

Summary and Discussion

Data-driven workflows provide the backbone of many important applications such as e-commerce, business process management, scientific applications and healthcare, for which formal verification is critically important. In addition, the proliferation of tools supporting high-level specification of the workflows provides natural targets for verification.

This thesis studies data-driven workflows verification in both theory and practice. From the theory perspective, the thesis shows decidability of verification for HAS, a rich artifact system model capturing core elements of IBM's successful GSM model. Leveraging novel techniques, including a hierarchy of Vector Addition Systems (or equivalently Petri nets) and a variant of quantifier elimination, we showed new complexity upper bounds for verification, ranging from PSPACE to non-elementary in various cases. Compared to previous work [36] where the best upper bound is non-elementary in the number of variables when database keys and/or arithmetic are present, our new upper bounds are elementary (EXPSPACE to 3EXPSPACE) for acyclic and linearly-cyclic schemas even in the presence of arithmetic and artifact relations. Moreover, even for arbitrary cyclic schema, the non-elementary complexity is only in the height of the task hierarchy, likely to be much smaller than the number of variables in the workflow specification.

From the practical perspective, the thesis presents two successful implementations: SpinArt and VERIFAS. Using the widely used off-the-shelf model checker Spin and a set of nontrivial optimizations, SpinArt achieves good performance on a realistic set of data-driven business processes. VERIFAS pushed further the frontier of practical verification with a specialized implementation built from scratch. VERIFAS uses the HAS* model, a variant of HAS that strikes a more practically relevant trade-off between expressivity and verification complexity. In particular, we demonstrate its practical applicability by showing that it is sufficiently expressive to specify a benchmark of realistic business processes extended with data-aware features. By adapting the symbolic verification techniques developed for HAS, coupled with the classic Karp-Miller algorithm accelerated with an array of nontrivial novel optimizations, VERIFAS achieves excellent performance not only on the real benchmark, but also on a synthetic set consisting of workflows of much larger size. Compared to SpinArt, VERIFAS not only applies to a broader class of data-driven workflows, but also has a decisive performance advantage. To our knowledge, SpinArt and VERIFAS are the first implementations of practical significance of workflow verifiers with full support of unbounded data.

Future work

There are several promising future work opportunities towards improving the applicability and impact of the presented results.

On the theoretical front, the quest for the right package of restrictions that enables verification while capturing more relevant sets of specifications is still ongoing. For example, real-life artifacts often require the ability to aggregate data collections (e.g. summing up all items in the shopping cart). While adding features like aggregate functions over the artifact relations can satisfy this need, it requires additional restrictions to the model and adaptation of the verification techniques. Other useful extensions involve checking properties of the interaction among multiple actors in data-driven workflows, or among multiple artifact instances evolving in parallel. The inter-operation, evolution, and integration of multiple systems also raise important static analysis questions.

Bridging the gap between the abstract setting of theoretical results and full-fledged specification frameworks also raises significant challenges. The current decidability results are subject to strong restrictions. The boundary of decidability is subtle, as even small deviations from the restrictions may lead to undecidability. This raises a need to design user interfaces for workflow developers to guide the development of full-fledged specifications towards satisfaction of the restrictions, whenever possible.

Clearly, a practical verifier needs to also deal with specifications that do not obey the restrictions needed for decidability. As typical in software verification, this can be done by abstracting the given specification to one that satisfies the restrictions, and verifying the resulting abstraction. For example, if certain arithmetic operations are not supported by the verifier, they can be abstracted as black-box relations, ignoring their semantics. The resulting verifier is guaranteed to be *sound* (it is never wrong when it claims correctness of a specification), but is possibly not *complete* (it may produce false negatives, i.e. candidate counterexamples to the desired property, which need to be validated by the user). The technical challenge lies in automatically generating the abstraction such that it gives up only as little completeness as necessary for decidability. Also, in the case of false negatives, how user feedback can guide changing the abstraction is an interesting and technically challenging research question.

Finally, the results and implementations presented in this thesis have boarder relevance beyond data-driven workflows. One promising impact area is the verification of blockchain-enabled applications specified using smart contracts. The blockchain paradigm is being adopted in support of new platforms for business collaboration in various areas, including finance, supply chain, food production, pharmaceuticals and healthcare [75]. The key component enabling such functionality is blockchain's programmable logic in the form of smart contracts. Essentially, similar to data-driven workflows, a smart contract specifies the business data stored in the blockchain and the transaction logic for updating the data invoked by external calls to the contract. Due to the high cost of vulnerabilities in smart contracts (e.g. the \$50 million loss caused by the DAO hack [113]), formal verification has become a highly desirable features in major blockchain platforms [73]. However, the commonly used smart contract languages are general-purpose, which means that automatic verification is not feasible in general. Therefore, a new high-level language for smart contracts (such as those presented in this thesis) not only enables efficient development, but also renders verification more feasible. Development of such a language is underway[1], and the new features specific to blockchain and smart contracts will bring new verification challenges. The techniques developed in this thesis provide a powerful toolbox for addressing them.

[1]The thesis author was involved in an ongoing related project during an internship in IBM Research.

Bibliography

[1] Object management group business process model and notation. http://www.bpmn.org/. Accessed: 2017-03-01.

[2] Web Ratio. http://www.webratio.com/.

[3] MIST - a safety checker for Petri Nets and extensions. https://github.com/pierreganty/mist/wiki, 2017.

[4] Parosh Aziz Abdulla, C. Aiswarya, Mohamed Faouzi Atig, Marco Montali, and Othmane Rezine. Recency-bounded verification of dynamic database-driven systems. In *PODS*, pages 195–210, 2016.

[5] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000.

[6] Eric Badouel, Loïc Hélouët, and Christophe Morvan. Petri nets with semi-structured data. In *Petri Nets*, 2015.

[7] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[8] Saugata Basu, Richard Pollak, and Marie-Françoise Roy. On the number of cells defined by a family of polynomials on a variety. *Mathematika*, 43(1):120–126, 1996.

[9] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. A computationally-grounded semantics for artifact-centric systems and abstraction results. In *IJCAI*, pages 738–743, 2011.

[10] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of deployed artifact systems via data abstraction. In *ICSOC*, 2011.

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

[12] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of gsm-based artifact-centric systems through finite abstraction. In *ICSOC*, 2012.

[13] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of agent-based artifact systems. *J. Artif. Intell. Res. (JAIR)*, 51:333–376, 2014.

[14] Kamal Bhattacharya, Nathan S Caswell, Santhosh Kumaran, Anil Nigam, and Frederick Y Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.

[15] Kamal Bhattacharya, Cagdas Gerede, Richard Hull, Rong Liu, and Jianwen Su. Towards formal analysis of artifact-centric business process models. In *International Conference on Business Process Management*, pages 288–304. Springer, 2007.

[16] Kamal Bhattacharya, Robert Guttman, Kelly Lyman, Fenno F Heath III, Santhosh Kumaran, Prabir Nandi, Frederick Wu, Prasanna Athma, Christoph Freiberg, Lars Johannsen, et al. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Systems Journal*, 44(1):145–162, 2005.

[17] BizAgi and Cordys and IBM and Oracle and Singularity and SAP AG and Agile Enterprise Design and Stiftelsen SINTEF and TIBCO and Trisotech. Case Management Model and Notation (CMMN), FTF Beta 1, Jan. 2013. OMG Document Number dtc/2013-01-01, Object Management Group.

[18] Michel Blockelet and Sylvain Schmitz. Model checking coverability graphs of vector addition systems. In *Mathematical Foundations of Computer Science 2011*, pages 108–119. Springer, 2011.

[19] D. Boaz, L. Limonad, and M. Gupta. BizArtifact: Artifact-centric Business Process Management, June 2013. `http://sourceforge.net/projects/bizartifact/`.

[20] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS*, pages 7–16. IEEE, 2006.

[21] Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. Rewriting systems with data. In *International Symposium on Fundamentals of Computation Theory*, pages 1–22. Springer, 2007.

[22] Ahmed Bouajjani, Peter Habermehl, and Richard Mayr. Automatic verification of recursive procedures with one integer parameter. *Theoretical Computer Science*, 295(1-3):85–106, 2003.

[23] Ahmed Bouajjani, Yan Jurski, and Mihaela Sighireanu. A generic framework for reasoning about dynamic networks of infinite-state processes. In *TACAS*, pages 690–705. Springer, 2007.

[24] Patricia Bouyer. A logical characterization of data languages. *Information Processing Letters*, 84(2):75–85, 2002.

[25] Patricia Bouyer, Antoine Petit, and Denis Thérien. An algebraic approach to data languages and timed languages. *Information and Computation*, 182(2):137–162, 2003.

[26] Marco Brambilla, Stefano Ceri, Sara Comai, Piero Fraternali, and Ioana Manolescu. Specification and design of workflow-driven hypertexts. *Journal of Web Engineering*, 1(2):163–182, 2002.

[27] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. of the ACM*, 30(2):323–342, 1983.

[28] Olaf Burkart, Didier Caucal, Faron Moller, and Bernhard Steffen. Verification on infinite structures. In *Handbook of Process algebra*, pages 545–623. Elsevier, 2001.

[29] Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Jianwen Su. Artifact-centric workflow dominance. In *Service-Oriented Computing*, pages 130–143. Springer, 2009.

[30] Diego Calvanese, Giuseppe De Giacomo, and Marco Montali. Foundations of data-aware process analysis: a database theory perspective. In *PODS*, 2013.

[31] Diego Calvanese, Giorgio Delzanno, and Marco Montali. Verification of relational multiagent systems with data types. In *AAAI*, pages 2031–2037, 2015.

[32] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing data-intensive Web applications*. Morgan-Kaufmann, 2002.

[33] Tian Chao, David Cohn, Adrian Flatgard, Sandy Hahn, Mark Linehan, Prabir Nandi, Anil Nigam, Florian Pinel, John Vergo, and Frederick y Wu. Artifact-based transformation of ibm global financing. In *International Conference on Business Process Management*, pages 261–277. Springer, 2009.

[34] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[35] David Cohn, Pankaj Dhoolia, Fenno Heath, Florian Pinel, and John Vergo. Siena: From powerpoint to web app in 5 minutes. In *ICSOC*, 2008.

[36] Elio Damaggio, Alin Deutsch, and Victor Vianu. Artifact systems with data dependencies and arithmetic. *ACM Transactions on Database Systems (TODS)*, 37(3):22, 2012. Also in ICDT 2011.

[37] Elio Damaggio, Richard Hull, and Roman Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with guard–stage–milestone lifecycles. *Information Systems*, 38(4):561–584, 2013.

[38] Giuseppe De Giacomo, Riccardo De Masellis, and Riccardo Rosati. Verification of conjunctive artifact-centric services. *Int. J. Cooperative Inf. Syst.*, 21(2):111–140, 2012.

[39] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. AAAI Press, 2013.

[40] Henk de Man. Case management: Cordys approach, 2009.

[41] Riccardo De Masellis, Chiara Di Francescomarino, Chiara Ghidini, Marco Montali, and Sergio Tessaris. Add data into business process verification: Bridging the gap between theory and practice. In *AAAI*, 2017.

[42] Stéphane Demri and Ranko Lazić. Ltl with the freeze quantifier and register automata. volume 10, page 16. ACM, 2009.

[43] Stéphane Demri, Ranko Lazić, and Arnaud Sangnier. Model checking freeze ltl over one-counter automata. In *International Conference on Foundations of Software Science and Computational Structures*, pages 490–504. Springer, 2008.

[44] Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *ICDT*, pages 252–267. ACM, 2009.

[45] Alin Deutsch, Yuliang Li, and Victor Vianu. Verification of hierarchical artifact systems. In *PODS*, pages 179–194, 2016.

[46] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD*, pages 539–550, 2005.

[47] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD*, pages 539–550. ACM, 2005.

[48] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web services. In *PODS*, pages 71–82. ACM, 2004.

[49] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web services. *JCSS*, 73(3):442–474, 2007.

[50] Alin Deutsch, Liying Sui, Victor Vianu, and Dayou Zhou. A system for specification and verification of interactive, data-driven web applications. In *SIGMOD*, pages 772–774, 2006.

[51] Alin Deutsch, Liying Sui, Victor Vianu, and Dayou Zhou. A system for specification and verification of interactive, data-driven web applications. In *SIGMOD*, pages 772–774. ACM, 2006.

[52] Alin Deutsch, Liying Sui, Victor Vianu, and Dayou Zhou. Verification of communicating data-driven web services. In *PODS*, pages 90–99. ACM, 2006.

[53] Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35(4):413–422, 1913.

[54] Volker Diekert and Paul Gastin. Pure future local temporal logics are expressively complete for Mazurkiewicz traces. In *LATIN 2004: Theoretical Informatics, 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004, Proceedings*, pages 232–241, 2004.

[55] Volker Diekert and Paul Gastin. Pure future local temporal logics are expressively complete for Mazurkiewicz traces. *Inf. Comput.*, 204(11):1597–1619, 2006.

[56] Volker Diekert and Grzegorz Rozenberg. *The book of traces*. World scientific, 1995.

[57] Guozhu Dong, Richard Hull, Bharat Kumar, Jianwen Su, and Gang Zhou. A framework for optimizing distributed workflow executions. In *DBPL*, pages 152–167. Springer, 1999.

[58] E. Allen Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[59] Paul Erdos. Graph theory and probability. *canad. J. Math*, 11(11):34–38, 1959.

[60] Expedia.Com, 2016.

[61] Alain Finkel. The minimal coverability graph for Petri nets. *Advances in Petri Nets 1993*, pages 210–243, 1993.

[62] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set of petri nets. *International Journal of Foundations of Computer Science*, 21(02):135–165, 2010.

[63] Cagdas E Gerede, Kamal Bhattacharya, and Jianwen Su. Static analysis of business artifact-centric operational models. In *SOCA*, pages 133–140. IEEE, 2007.

[64] Cagdas E Gerede and Jianwen Su. Specification and verification of artifact behaviors in business process models. In *ICSOC*, pages 181–192. Springer, 2007.

[65] Robert J Glushko and Tim McGrath. *Document Engineering: Analyzing and Designing Documents for Business Informatics and Web Services MIT Press*. 2005.

[66] Pavel Gonzalez, Andreas Griesmayer, and Alessio Lomuscio. Verifying GSM-based business artifacts. In *International Conference on Web Services (ICWS)*, pages 25–32, 2012.

[67] Pavel Gonzalez, Andreas Griesmayer, and Alessio Lomuscio. Model checking gsm-based multi-agent systems. In *ICSOC*, pages 54–68, 2013.

[68] Pavel Gonzalez, Andreas Griesmayer, and Alessio Lomuscio. Verification of gsm-based artifact-centric systems by predicate abstraction. In *ICSOC*, pages 253–268, 2015.

[69] Peter Habermehl. On the complexity of the linear-time μ-calculus for petri nets. In *Application and Theory of Petri Nets 1997*, pages 102–116. Springer, 1997.

[70] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Riccardo De Masellis, and Paolo Felli. Foundations of relational artifacts verification. In *BPM*, pages 379–395. Springer, 2011.

[71] Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. In *PODS*, 2013.

[72] Joos Heintz, Pablo Solernó, and Marie-Françoise Roy. On the complexity of semialgebraic sets. In *IFIP Congress*, pages 293–298, 1989.

[73] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[74] Gerard Holzmann. *Spin Model Checker, The: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[75] Richard Hull, Vishal S Batra, Yi-Min Chen, Alin Deutsch, Fenno F Terry Heath III, and Victor Vianu. Towards a shared ledger business collaboration language based on data-aware processes. In *International Conference on Service-Oriented Computing*, pages 18–36. Springer, 2016.

[76] Richard Hull, Elio Damaggio, Riccardo De Masellis, Fabiana Fournier, Manmohan Gupta, Fenno Terry Heath III, Stacy Hobson, Mark Linehan, Sridhar Maradugu, Anil Nigam, et al. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *DEBS*, pages 51–62. ACM, 2011.

[77] Richard Hull, Francois Llirbat, Bharat Kumar, Gang Zhou, Guozhu Dong, and Jianwen Su. Optimization techniques for data intensive decision flows. In *ICDE*, 2000.

[78] Richard Hull, Francois Llirbat, Eric Siman, Jianwen Su, Guozhu Dong, Bharat Kumar, and Gang Zhou. Declarative workflows that support easy modification and dynamic browsing. volume 24, pages 69–78. ACM, 1999.

[79] Richard Hull and Jianwen Su. Tools for design of composite web services. In *SIGMOD*, pages 958–961. ACM, 2004.

[80] Richard Hull, Jianwen Su, and Roman Vaculín. Data management perspectives on business process management: tutorial overview. In *SIGMOD*, 2013.

[81] Marcin Jurdzinski and Ranko Lazic. Alternation-free modal mu-calculus for data trees. In *LICS*, pages 131–140. IEEE, 2007.

[82] H.W. Kamp. Tense logic and the theory of linear order, 1968. Phd thesis, University of California, Los Angeles.

[83] Richard M Karp, Raymond E Miller, and Arnold L Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 125–136. ACM, 1972.

[84] Ralph Kimball and Margy Ross. The data warehouse toolkit: the complete guide to dimensional modeling. 2011.

[85] Santhosh Kumaran, Prabir Nandi, Terry Heath, Kumar Bhaskaran, and Raja Das. Adoc-oriented programming. In *Symposium on Applications and the Internet*, pages 334–341. IEEE, 2003.

[86] Ranko Lazić, Tom Newcomb, Joël Ouaknine, Andrew W Roscoe, and James Worrell. Nets with tokens which carry data. volume 88, pages 251–274. IOS Press, 2008.

[87] Ranko Lazić, Tom Newcomb, Joël Ouaknine, Andrew W Roscoe, and James Worrell. Nets with tokens which carry data. *Fundamenta Informaticae*, 88(3):251–274, 2008.

[88] Yuliang Li, Alin Deutsch, and Victor Vianu. Spinart: A spin-based verifier for artifact systems. *arXiv preprint*, arXiv:1705.09427, 2017.

[89] Yuliang Li, Alin Deutsch, and Victor Vianu. Verifas: a practical verifier for artifact systems. *Proceedings of the VLDB Endowment*, 11(3):283–296, 2017.

[90] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[91] Richard Lipton. The reachability problem requires exponential space. *Research Report 62, Department of Computer Science, Yale University, New Haven, Connecticut*, 1976.

[92] Rong Liu, Kamal Bhattacharya, and Frederick Y Wu. Modeling business contexture and behavior using business artifacts. In *International Conference on Advanced Information Systems Engineering*, pages 324–339. Springer, 2007.

[93] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[94] Mike Marin, Richard Hull, and Roman Vaculín. Data centric bpm and the emerging case management standard: A short survey. In *BPM Workshops*, 2012.

[95] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, et al. Owl-s: Semantic markup for web services, 2004.

[96] Richard Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297(1):337–354, 2003.

[97] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. *DAIMI Report Series*, 6(78), 1977.

[98] Sheila A McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE intelligent systems*, 16(2):46–53, 2001.

[99] Stephan Merz. Model checking: A tutorial overview. In *Summer School on Modeling and Verification of Parallel Processes*, pages 3–38. Springer, 2000.

[100] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, 1967.

[101] Prabir Nandi and Santhosh Kumaran. Adaptive business objects-a new component model for business integration. In *ICEIS (3)*, pages 179–188, 2005.

[102] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3):403–435, 2004.

[103] Anil Nigam and Nathan S Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[104] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

[105] Amir Pnueli. The temporal logic of programs. In *FOCS*, 1977.

[106] E. L. Post. Recursive unsolvability of a problem of Thue. *J. of Symbolic Logic*, 12:1–11, 1947.

[107] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.

[108] Pierre-Alain Reynier and Frédéric Servais. Minimal coverability set for Petri nets: Karp and Miller algorithm with pruning. In *Int'l. Conf. on Application and Theory of Petri Nets and Concurrency*, pages 69–88. Springer, 2011.

[109] Ronald L Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[110] Fernando Rosa-Velardo and David de Frutos-Escrig. Decidability and complexity of Petri nets with unordered data. *Theoretical Computer Science*, 412(34):4439–4451, 2011.

[111] Luc Segoufin and Szymon Toruńczyk. Automata based verification over linearly ordered data domains. In *STACS*, volume 9, pages 81–92, 2011.

[112] Natalia Sidorova, Christian Stahl, and Nikola Trčka. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Information Systems*, 36(7):1026–1043, 2011.

[113] David Siegel. Understanding the dao attack. *Web. http://www. coindesk. com/understanding-dao-hack-journalists*, 2016.

[114] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill, 2005.

[115] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[116] A Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, 1994.

[117] A Prasad Sistla and Edmund M Clarke. The complexity of propositional linear temporal logics. *JACM*, 32(3):733–749, 1985.

[118] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[119] Dmitry Solomakhin, Marco Montali, Sergio Tessaris, and Riccardo De Masellis. Verification of artifact-centric systems: Decidability and modeling issues. In *ICSOC*, pages 252–266, 2013.

[120] Marc Spielmann. Verification of relational transducers for electronic commerce. *JCSS*, 66(1):40–65, 2003.

[121] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[122] Alfred Tarski. A decision method for elementary algebra and geometry. *1948*, 1951.

[123] Wil MP Van Der Aalst. Business process management: a comprehensive survey. *ISRN Software Engineering*, 2013.

[124] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 322–331, 1986.

[125] Moshe Y Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and computation*, 115(1):1–37, 1994.

[126] Panos Vassiliadis and Timos Sellis. A survey of logical models for olap databases. *ACM Sigmod Record*, 28(4):64–69, 1999.

[127] Jianrui Wang and Akhil Kumar. A framework for document-driven workflow systems. In *BPM*, pages 285–301. Springer, 2005.

[128] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

[129] Xiangpeng Zhao, Jianwen Su, Hongli Yang, and Zongyan Qiu. Enforcing constraints on life cycles of business artifacts. In *Theoretical Aspects of Software Engineering*, pages 111–118. IEEE, 2009.

[130] Wei-Dong Zhu, Brian Benoit, Bob Jackson, Johnson Liu, Mike Marin, Seema Meena, Juan Felipe Ospina, Guillermo Rios, et al. Advanced case management with ibm case manager, 2015.