Verification of Hierarchical Artifact Systems

Alin Deutsch, Yuliang Li, Victor Vianu

Verification of Hierarchical Artifact Systems

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

Yuliang Li
UC San Diego
yul206@eng.ucsd.edu

Victor Vianu
UC San Diego & INRIA-Saclay
vianu@cs.ucsd.edu

ABSTRACT

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. The present work represents a significant advance on the problem of artifact verification, by considering a much richer and more realistic model than in previous work, incorporating core elements of IBM's successful Guard-Stage-Milestone model. In particular, the 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, making use of novel techniques including a hierarchy of Vector Addition Systems and a variant of quantifier elimination tailored to our context.

Keywords

data-centric workflows; business process management; temporal logic; verification

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 [40], successfully deployed in practice [10, 9, 17, 22, 52] 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 (see related work).

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) [19, 34]. 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) [12, 37].

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, we have embarked upon a study of the verification problem for artifact systems. Rather than relying on general-purpose software verification tools suffering from well-known limitations, our aim 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. Our approach relies critically on the declarative nature of service specifications and brings into play a novel marriage of database and computer-aided verification techniques.

In previous work [23, 18], we studied the verification problem 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, we considered the problem of statically checking 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, we developed in [23] a symbolic approach allowing 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 [18] we extended our approach 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 work represents 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) [13]. 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 *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 we 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 [18], 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 1 (without arithmetic) and 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 [35, 50], 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 [18],

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).

The paper is organized as follows. The HAS model is presented in Section 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. An example HAS modeling a simple travel booking process is provided in the appendix. The temporal logic HLTL-FO is introduced in Section 3, together with a corresponding extension of Büchi automata to trees of local runs. In Section 4 we prove the decidability of verification 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 5 we show how the verification results can be extended in the presence of arithmetic. Section 6 traces the boundary of decidability, showing that the main restrictions adopted in defining the HAS model cannot be relaxed. Finally, we discuss related work in Section 7 and conclude. More details and proofs are provided in the extended appendix of the full version [25] of this paper.

2. FRAMEWORK

In this section we present the syntax and semantics of Hierarchical Artifact Systems (HAS's). We begin with the underlying database schema.

DEFINITION 1. *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 $\{\mathrm{ID}, F_1, \ldots, F_m\}$.*

To each foreign key attribute F_i of R is associated a relation R_{F_i} of $\mathcal{DB}$ and the inclusion dependency $R[F_i] \subseteq R_{F_i}[\mathrm{ID}]$. It is said that F_i references R_{F_i}.

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', $Dom(R.ID) \cap Dom(R'.ID) = \emptyset$. The domain of a foreign key attribute F referencing R is $Dom(R.ID)$. We denote by $DOM_{id} = \cup_{R \in \mathcal{DB}} Dom(R.ID)$. 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 as 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 of 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 D, denoted $\mathtt{adom}(D)$, 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* if the graph FK is acyclic, and it is *linearly-cyclic* if each relation R is contained in at most one simple cycle.

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 is numeric is also harmless. Indeed, an 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 [18]. 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 6.

Similarly to the database schema, we consider two infinite, disjoint sets VAR_{id} of ID variables and $VAR_{\mathbb{R}}$ of numeric variables. We associate to each variable x its *domain* $Dom(x)$. If $x \in VAR_{id}$, then $Dom(x) = \{\texttt{null}\} \cup DOM_{id}$, where $\texttt{null} \notin DOM_{id} \cup \mathbb{R}$ ($\texttt{null}$ plays a special role that will become clear shortly). If $x \in VAR_{\mathbb{R}}$, then $Dom(x) = \mathbb{R}$. An *artifact variable* is a variable in $VAR_{id} \cup VAR_{\mathbb{R}}$. 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 2. *A **task schema** over database schema $\mathcal{DB}$ is a triple $T = \langle \bar{x}^T, S^T, \bar{s}^T \rangle$ 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, and $\bar{s}^T$ is a sequence of k distinct id variables in $\bar{x}^T$.*

We denote by $\bar{x}_{id}^T = \bar{x}^T \cap VAR_{id}$ and $\bar{x}_{\mathbb{R}}^T = \bar{x}^T \cap VAR_{\mathbb{R}}$. We refer to S^T as the *artifact relation* or *set* of T.

DEFINITION 3. *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 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.

DEFINITION 4. *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 a finite instance of $\mathcal{DB}$, $\bar{S}$ a finite 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 $\{\texttt{init}, \texttt{active}, \texttt{closed}\}$.*

The stage $stg(T_i)$ of a task T_i has the following intuitive meaning in the context of a run of its parent: $\texttt{init}$ indicates that T_i has not yet been called within the run, $\texttt{active}$ says that T_i has been called and has not returned its answer, and $\texttt{closed}$ indicates that T_i has returned its answer. As we will see, a task T_i can only be called once within a given run of

its parent. However, it can be called again in subsequent runs.

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 $\texttt{null}$ 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 $R(ID, A_1, \ldots, A_m, F_1, \ldots, F_n) \in \mathcal{DB}$, $\{x, z_1, \ldots, z_n\} \subseteq VAR_{id}$ and $\{y_1, \ldots, y_m\} \subseteq VAR_{\mathbb{R}}$. 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.

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

DEFINITION 5. *Let $T = \langle \bar{x}^T, S^T, \bar{s}^T \rangle$ be a task 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 set updates; $+S^T(\bar{s}^T)$ and $-S^T(\bar{s}^T)$ are called the **insertion** and **retrieval** of $\bar{s}^T$, respectively.*

Intuitively, $+S^T(\bar{s}^T)$ 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 current 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 (see definition of the semantics below).

As will become apparent, although pre-and-post conditions are quantifier-free, $\exists$FO conditions can be simulated by adding variables to $\bar{x}^T$.

An internal service of a task T specifies transitions that only modify the variables $\bar{x}^T$ of T and the contents of S^T. Interactions among tasks are specified using two kinds of special services, called the *opening-services* and *closing-services*.

DEFINITION 6. *Let T_c be a child of a task T in $\mathcal{A}$.*
*(i) The **opening-service** $\sigma_{T_c}^o$ of T_c is a tuple $\langle \pi, f_{in} \rangle$, where π is a condition over $\bar{x}^T$, and f_{in} is a partial 1-1 mapping from $\bar{x}^{T_c}$ to $\bar{x}^T$ (called the input variable mapping). We denote $dom(f_{in})$ by $\bar{x}_{in}^{T_c}$, called the **input variables** of T_c, and $range(f_{in})$ by $\bar{x}_{T_c \downarrow}^T$ (the variables of T passed as input to T_c).*
*(ii) The **closing-service** $\sigma_{T_c}^c$ of T_c is a tuple $\langle \pi, f_{out} \rangle$, where π is a condition over $\bar{x}^{T_c}$, and f_{out} is a partial 1-1 mapping from $\bar{x}^T$ to $\bar{x}^{T_c}$ (called the output variable mapping). We denote $dom(f_{out})$ by $\bar{x}_{T_c \uparrow}^T$, referred to as the **returned variables** from T_c. It is required that $\bar{x}_{T_c \uparrow}^T \cap \bar{x}_{in}^T = \emptyset$. We denote by $\bar{x}_{ret}^{T_c}$ the **to-be-returned variables** (or return variables), defined as $range(f_{out})$.*

Intuitively, the opening-service $\langle \pi, f_{in} \rangle$ of a task T_c specifies the condition π that the parent task T has to satisfy

in order to open T_c. When T_c is opened, a subset of the variables of T are sent to T_c according to the mapping f_{in}. Similarly, the closing-service $\langle \pi, f_{out} \rangle$ specifies the condition π that T_c has to satisfy in order to be closed and return to T. When T_c is closed, a subset of $\bar{x}^{T_c}$ is sent back to T, as specified by f_{out}.

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 $\bar{x}^T$.

DEFINITION 7. *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 over* $\mathcal{A}$ *including* σ^o_T *and* σ^c_T *for each task* T *of* $\mathcal{A}$*, and* Π *is a condition over* $\bar{x}^{T_1}_{in}$ *(where* T_1 *is the root task).*

We next define the semantics of HAS's. 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. 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 $\bar{x}^T$ is 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 a subset of $\bar{x}^{T_c}$ are sent to T as T's returned variables from T_c. An internal service of T can only be applied after all active subtasks of T have returned their answer.

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's 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 8. *Let* $T = \langle \bar{x}^T, S^T, \bar{s}^T \rangle$ *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* (π, ψ)*, then:*

- $D \cup \mathcal{C} \models \pi(\nu)$ *and* $D \cup \mathcal{C} \models \psi(\nu')$

- $\nu'(y) = \nu(y)$ *for each* y *in* $\bar{x}^T_{in}$
- *if* $\delta = \{+S^T(\bar{s}^T)\}$*, then* $S' = S \cup \{\nu(\bar{s}^T)\}$*,*
- *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* $\nu(z) \neq \mathtt{null}$*. Finally, if* $\sigma = \sigma^c_T$ *then* $I' = I$*.*

We now define local runs.

DEFINITION 9. *Let* $T = \langle \bar{x}^T, S^T, \bar{s}^T \rangle$ *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* $i \ge 0$*,* I_i *is an instance of* T *and* $\sigma_i \in \Sigma^{obs}_T$
- ν_{in} *is a valuation of* $\bar{x}^T_{in}$
- $\sigma_0 = \sigma^o_T$ *and* $S_0 = \emptyset$*,*
- $\nu_0|\bar{x}^T_{in} = \nu_{in}$*,* $\nu_0(z) = \mathtt{null}$ *for* $z \in \mathrm{VAR}_{id} - \bar{x}^T_{in}$ *and* $\nu_0(z) = 0$ *for* $z \in \mathrm{VAR}_{\mathbb{R}} - \bar{x}^T_{in}$
- *if for some* i*,* $\sigma_i = \sigma^c_T$ *then* $\gamma \in \mathbb{N}$ *and* $i = \gamma - 1$ *(and* ρ_T *is called a* returning *local run)*
- $\nu_{out} = \nu_{\gamma-1}|\bar{x}^T_{ret}$ *if* ρ_T *is a returning run and* $\bot$ *otherwise*
- *a* segment *of* ρ_T *is a subsequence* $\{(I_i, \sigma_i)\}_{i \in J}$*, where* J *is a maximal interval* $[a, b] \subseteq \{i \mid 0 \le i < \gamma\}$ *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* $\sigma_{\gamma-1} = \sigma^c_T$ *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* $\sigma_i = \sigma^o_{T_c}$*. If* J *is not blocking 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 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* $\bar{x}^{T_1}_{in}$ *such that* $D \cup \mathcal{C} \models \Pi$*, and* $\nu_{out} = \bot$ *(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^{oc}_T 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 6.

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 10. *A* tree of local runs *is a directed labeled tree* **Tree** *in which each node is 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* $\rho_T = (\nu^T_{in}, \nu^T_{out}, \{(I_i, \sigma_i)\}_{0 \le i < \gamma})$ *be a node of* **Tree***, where* $I_i = (\nu_i, S_i)$*,* $i \ge 0$*. 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* ρ_T *to a node* $\rho_{T_c} = (\nu_{in}, \nu_{out}, \{(I'_i, \sigma'_i)\}_{0 \le i < \gamma'})$ *of* **Tree***, and the following hold:*

- $\nu_{in} = f_{in} \circ \nu_i$ *where[1]* f_{in} *is the input variable mapping of* $\sigma^o_{T_c}$

---

[1] Composition is left-to-right.

- ρ_{T_c} *is a returning run iff there exists* $j > i$ *such that* $\sigma_j = \sigma_{T_c}^c$; *let* k *be the minimum such* j. *Then* $\nu_k(z) = \nu_{out}(f_{out}(z))$ *for every* $z \in \bar{x}_{T_c\uparrow}^T$ *for which* $\nu_{k-1}(z) = \texttt{null}$, *where* f_{out} *is the output mapping of* $\sigma_{T_c}^c$.

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 infinite or blocking).*

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.

We next turn to global runs. 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 (see Appendix B.1 of [25] for the formal definition). For a tree of local runs **Tree**, we denote by $\mathcal{L}(\textbf{Tree})$ the set of all global runs induced by the legal interleavings of **Tree**.

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 **X** (next), **U** (until), **G** (always) and **F** (eventually) (e.g., see [29]). An extension of LTL in which propositions are interpreted as FO sentences has previously been defined to specify properties of sequences of structures [47], and in particular of runs of artifact systems [23, 18]. The extension is denoted by LTL-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.

THEOREM 11. *It is undecidable, given an LTL-FO formula* φ *and a HAS* $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$, *whether* $\Gamma \models \varphi$. *Moreover, this holds even for LTL formulas over* Σ *(restricting the sequence of services in a global run).*

The proof, provided in Appendix B.3 of [25], is by reduction from repeated state reachability in VASS with resets and bounded lossiness, whose undecidability follows from [38]. .

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* [41]. Moreover, HLTL-FO enjoys a pleasing completeness property: it expresses, in a reasonable sense, *all* interleaving-invariant LTL-FO properties of HAS's. The proof is non-trivial, building on completeness results for propositional temporal logics on Mazurkiewicz traces [27, 28] (more details are provided in Appendix B.4 of [25]).

We next define HLTL-FO. Propositions in HLTL-FO are interpreted as conditions[2] on artifact instances in the run, or recursively as HLTL-FO formulas on runs of invoked children tasks. The different conditions may share some universally quantified global variables.

DEFINITION 12. *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* $\text{VAR}_{id} \cup \text{VAR}_{\mathbb{R}}$ *disjoint from* $\{\bar{x}^T \mid T \in \mathcal{H}\}$, *called* global variables. *We first define recursively the set* $\Psi(T, \bar{y})$ *of* basic HLTL-FO formulas *with global variables* $\bar{y}$, *for each task* $T \in \mathcal{H}$. *The set* $\Psi(T, \bar{y})$ *consists of all formulas* φ_f *obtained as follows:*

- φ *is an LTL formula with propositions* $P \cup \Sigma_T^{obs}$ *where* P *is a finite set of proposition disjoint from* Σ;
- *Let* Φ *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* $\bar{y} \cap \text{VAR}_{id}$; f *is a function from* P *to*[3] $\Phi \cup \{[\psi]_{T_c} \mid \psi \in \Psi(T_c, \bar{y}), T_c \in child(T)\}$;
- φ_f *is obtained by replacing each* $p \in P$ *with* $f(p)$;

An HLTL-FO formula over $\mathcal{A}$ *is an expression* $\forall \bar{y}[\varphi_f]_{T_1}$ *where* φ_f *is in* $\Psi(T_1, \bar{y})$.

In an HLTL-FO formula of task T, each proposition is mapped to either a quantifier-free FO formula referring to the variables and set of task T, or an HLTL-FO formula of a child task of T. The intuition is the following. A proposition mapped to a quantifier-free FO formula holds in a given configuration of T if the formula is true in that configuration. A proposition mapped to an expression $[\psi]_{T_c}$ 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.

EXAMPLE 13. *Let* T_1 *be a root task with child tasks* T_2 *and* T_3. *The HLTL-FO formula (with no global variables)*

$$\varphi = [\ \mathbf{F}[\psi_2]_{T_2} \rightarrow \mathbf{G}(\sigma_{T_3}^o \rightarrow [\psi_3]_{T_3})]_{T_1}$$

states that whenever T_1 *calls child task* T_2 *and* T_2's *local run satisfies property* ψ_2, *then if* T_3 *is also called (via the opening service* $\sigma_{T_3}^o$), *its local run must satisfy property* ψ_3.

See Appendix A.2 for a concrete HLTL-FO property of similar structure, in the context of our example for the HAS model.

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 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 a formula in $\Psi(T, \langle \rangle)$ (no global variables). Recall that φ is a propositional LTL formula over $P \cup \Sigma_T^{obs}$. Let $\rho_T = (\nu_{in}, \nu_{out}, \{(I_i, \sigma_i)\}_{i < \gamma})$ be a local run of T in **Tree**. A proposition $\sigma \in \Sigma_T^{obs}$ holds in (I_j, σ_j) if $\sigma = \sigma_j$. Consider $p \in P$ and $f(p)$. If $f(p)$ is an FO formula, the standard definition applies. If $f(p) = [\psi]_{T_c}$, then (I_j, σ_j) satisfies $[\psi]_{T_c}$ iff $\sigma_j = \sigma_{T_c}^0$ and the local run of T_c connected to ρ_T in **Tree** by an edge labeled j satisfies ψ. The formula φ_f is satisfied if the sequence of 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 [21].

---

[2] For consistency with previous notation, we denote the logic HLTL-FO although the FO interpretations are restricted to be quantifier free.

[3] $[\psi]_{T_c}$ is an expression whose meaning is explained below.

A full tree of local runs satisfies $\varphi_f \in \Psi(T_1, \langle\rangle)$ if its root (a local run of T_1) satisfies φ_f. Finally, let $\varphi_f(\bar{y})$ be a formula in $\Psi(T_1, \bar{y})$. Then $\forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$ is satisfied by **Tree**, denoted **Tree** $\models \forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$, 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 $p \in P$. Note that $\varphi_{f^\nu} \in \Psi(T_1, \langle\rangle)$. Finally, Γ satisfies $\forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$, denoted $\Gamma \models \forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$, if **Tree** $\models \forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$ 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 $\forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$ be an HLTL-FO formula and $\rho \in \mathcal{L}(\textbf{Tree})$, where **Tree** is a full tree of local runs of Γ. We say that ρ satisfies $\forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$ if **Tree** satisfies $\forall \bar{y}[\varphi_f(\bar{y})]_{T_1}$. This is well defined in view of the following easily shown fact: if $\rho \in \mathcal{L}(\textbf{Tree}_1) \cap \mathcal{L}(\textbf{Tree}_2)$ then $\textbf{Tree}_1 = \textbf{Tree}_2$.

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. 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. More details and proofs of the above simplifications can be found in Appendix B.5 of [25]. 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 [49, 45]. 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 $\varphi = [\xi]_{T_1}$ 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 $\mathcal{B}_\varphi = \{B(T, \eta) \mid T \in \mathcal{H}, \eta \in 2^{\Phi_T}\}$.

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 $(\rho_j, \sigma_{T_c}^o)$ the truth assignment in $2^{\Phi_{T_c}}$ adorning its outgoing edge labeled j. (Recall that in configurations (I_j, σ_j) for which $\sigma_j \neq \sigma_{T_c}^o$, all formulas in Φ_{T_c} are *false* by definition.)
- $\alpha(\rho_{T_1})$ is accepted by the Büchi automaton B_ξ where $\alpha(\rho_{T_1})$ is defined as above.

The following can be shown.

LEMMA 14. *A full tree of local runs* **Tree** *satisfies* $\varphi = [\xi]_{T_1}$ *iff* **Tree** *is accepted by* $\mathcal{B}_\varphi$.

4. 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 5 how our approach can be extended when arithmetic is present.

The roadmap to verification is the following. Let Γ be a HAS and $\varphi = [\xi]_{T_1}$ an HLTL-FO formula over Γ. To verify that every tree of local runs of Γ satisfies φ, we check that there is no tree of local runs satisfying $\neg\varphi = [\neg\xi]_{T_1}$, or equivalently, accepted by $\mathcal{B}_{\neg\varphi}$. Since there are infinitely many trees of local runs of Γ due so 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 [23, 18], 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, 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) [13].

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 than available in the model. Moreover, Theorem 11 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 [18] is even less plausible, because of the lack of artifact relations. Note that, even if a reduction were possible, the results of [18] 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.

4.1 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. the 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 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.

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 $\xi_1.\xi_2.\ldots.\xi_m$, $\xi_1 = x_R$ for some $x \in \bar{x}_{id}^T$ and $R \in \mathcal{DB}$, ξ_j is a foreign key in some relation of $\mathcal{DB}$ for $2 \leq j < m$, ξ_m is a foreign key or a numeric attribute, ξ_2 is an attribute of R, and for each i, $2 < i \leq m$, if ξ_{i-1} is a foreign key referencing Q then ξ_i is an attribute of Q. We define the length of $\xi_1.\xi_2.\ldots.\xi_m$ as m. A *navigation set* $\mathcal{E}_T$ is a set of expressions such that:

- for each $x \in \bar{x}_{id}^T$ there is at most one $R \in \mathcal{DB}$ for which the expression x_R is in $\mathcal{E}_T$;
- every expression in $\mathcal{E}_T$ is of the form $x_R.w$ where $x_R \in \mathcal{E}_T$, and has length $\leq h(T)$;
- if $e \in \mathcal{E}_T$ then every expression $e.s$ of length $\leq h(T)$ extending e is also in $\mathcal{E}_T$.

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 $e \in \bar{x}_\mathbb{R}^T \cup \{0\}$ or $e = w.a$ where a is a numeric attribute; its sort is $\texttt{null}$ if $e = \texttt{null}$ or $e = x \in \bar{x}_{id}^T$ and $x_R \notin \mathcal{E}_T$ for all $R \in \mathcal{DB}$; and its sort is $\text{ID}(R)$ for $R \in \mathcal{DB}$ if $e = x_R$, or $e = x \in \bar{x}_{id}^T$ and $x_R \in \mathcal{E}_T$, or $e = w.f$ where f is a foreign key referencing R.

DEFINITION 15. *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* $e \sim_\tau f$ *then* e *and* f *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.

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 $x \sim_\tau y$,
- $R(x, y_1, \ldots, y_m, z_1, \ldots, z_n)$ holds in τ for relation $R(id, a_1, \ldots, a_m, f_1, \ldots, f_n)$ iff $\{x_R.a_1, \ldots, x_R.a_m, x_R.f_1, \ldots, x_R.f_n\} \subseteq \mathcal{E}_T$, and $(y_1, \ldots, y_m, z_1, \ldots, z_m) \sim_\tau (x_R.a_1, \ldots, x_R.a_m, x_R.f_1, \ldots, x_R.f_n)$
- Boolean combinations of conditions are standard.

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 $\bar{z}$ upto 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 $VAR_{id} \cup VAR_\mathbb{R}$ such that $f(\bar{x}_{id}^T) \subseteq VAR_{id}$, $f(\bar{x}_\mathbb{R}^T) \subseteq VAR_\mathbb{R}$ 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 seen above, 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, $(\bar{x}_{in}^T \cup \bar{s}^T)$. We refer to the projection of a T-isomorphism type onto $(\bar{x}_{in}^T \cup \bar{s}^T)$ 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 16. *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.

DEFINITION 17. *A local symbolic run $\tilde{\rho}_T$ of task T is a tuple $(\tau_{in}, \tau_{out}, \{(I_i, \sigma_i)\}_{0 \leq i < \gamma})$, where:*

- *each I_i is a symbolic instance $(\tau_i, \bar{c}_i)$ of T*
- *each σ_i is a service in Σ_T^{obs}*
- *$\gamma \in \mathbb{N} \cup \{\omega\}$ (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 $\bar{x}_{in}^T$. And $\tau_{in} \models \Pi$ if $T = T_1$.*
- *at the first instance I_0, $\tau_0|\bar{x}_{in}^T = \tau_{in}$, for every $x \in \bar{x}_{id}^T - \bar{x}_{in}^T$, $x \sim_{\tau_0} \texttt{null}$, and for every $x \in \bar{x}_\mathbb{R}^T - \bar{x}_{in}^T$, $x \sim_{\tau_0} 0$. Also $\bar{c}_0 = \bar{0}$ and $\sigma_0 = \sigma_T^o$.*

- *if for some i, $\sigma_i = \sigma_T^c$ then $\tilde{\rho}_T$ is finite and $i = \gamma - 1$ (and $\tilde{\rho}_T$ is called a* returning *run)*
- *τ_{out} is $\bot$ if $\tilde{\rho}_T$ is infinite or finite but $\sigma_{\gamma-1} \neq \sigma_T^c$, and it is $\tau_{\gamma-1}|(\bar{x}_{in}^T \cup \bar{x}_{ret}^T)$ 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 $\sigma_i = \sigma_{T_c}^o$. If J is not terminal and such i exists, there is exactly one $j \in J$ for which $\sigma_j = \sigma_{T_c}^c$, 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 $s \in \bar{s}^T$, $s \not\sim_{\hat{\tau}} \text{null}$ implies that there exists an expression $x_R.w$ in $\hat{\tau}$ such that $x \in \bar{x}_{in}^T$ and $x_R.w \sim_{\hat{\tau}} s$. We denote by $TS_{ib}(T)$ the set of input-bound types in $TS(T)$. 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 ($\bar{a}_0$ 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 $\delta^- = \{-S^T(\bar{s}^T)\}$.

Intuitively, 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. The input-bound TS-isomorphism types require special handling because consecutive insertions necessarily collide so the counter's value cannot go beyond 1.

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 $\hat{\tau} = \tau|(\bar{x}_{in}^T \cup \bar{s}^T)$ and $\hat{\tau}' = \tau'|(\bar{x}_{in}^T \cup \bar{s}^T)$,
  - $\tau|\bar{x}_{in}^T = \tau'|\bar{x}_{in}^T$,
  - $\tau \models \pi$ 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 $TS_{ib}(T)$.
- 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 $\bar{x}_{const}^T = \bar{x}^T - \{x \in \bar{x}_{T_c}^T | x \sim_\tau \text{null}\}$, $\tau'|\bar{x}_{const}^T = \tau|\bar{x}_{const}^T$ and $\bar{c}' = \bar{c}$.
- If σ' is the closing service $\sigma_T^c = \langle \pi, f_{out} \rangle$ of T, then $\tau \models \pi$ and $(\tau, \bar{c}) = (\tau', \bar{c}')$.

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 corresponding 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 20 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 20, is that only a bounded number of new tuples must be created, thus keeping the database finite.

DEFINITION 18. *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 $\sigma_i = \sigma_{T_c}^o$ 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:*
- *$\tau'_{in} = f_{in}^{-1}(\tau_i)|(\bar{x}_{in}^{T_c}, h(T_c))$ where f_{in} is the input variable mapping of $\sigma_{T_c}^o$*
- *$\tilde{\rho}_{T_c}$ is a returning run iff there exists $j > i$ such that $\sigma_j = \sigma_{T_c}^c$; let k be the minimum such j. Let $\bar{x}_r = \bar{x}_{T_c^\downarrow}^T$ and $\bar{x}_w = \{x | x \in \bar{x}_{T_c^\uparrow}^T, x \sim_{\tau_{k-1}} \text{null}\}$. Then $\tau_k|(\bar{x}_r \cup \bar{x}_w, h(T_c)) = ((f_{in} \circ f_{out}^{-1})(\tau_{out}))|(\bar{x}_r \cup \bar{x}_w)$ where f_{out} is the output variable mapping of $\sigma_{T_c}^c$.*

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 $\varphi = [\xi]_{T_1}$ over Γ. Satisfaction of φ 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 14, are also immediately extended to symbolic trees of runs. We state the following.

LEMMA 19. *A symbolic tree of runs* **Sym** *over Γ satisfies φ iff* **Sym** *is accepted by $\mathcal{B}_\varphi$.*

The key result enabling the use of symbolic trees of runs is the following (The proof is provided in the extended appendix of [25]).

THEOREM 20. *For an artifact system Γ and HLTL-FO property φ, 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, but the *if* part is non-trivial. The construction of an accepted tree of local runs from an accepted symbolic tree of runs **Sym** is done in two stages. First, an accepted tree of local runs over an *infinite* database is constructed, 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 and retrievals of tuples from S^T, and subject to satisfaction of the key constraints. In the second stage, the infinite database is turned into a finite one by carefully merging data values, while avoiding any inconsistencies.

4.2 Symbolic Verification

In view of Theorem 20, 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 be an HLTL-FO formula over Γ, and recall the automaton $\mathcal{B}_\varphi$ and associated notation from Section 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 $\boldsymbol{Sym}_T$ of $\mathcal{H}_T$ such that:

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

Note that there exists a symbolic tree of runs $\boldsymbol{Sym}$ over Γ satisfying $\varphi = [\xi]_{T_1}$ iff $(\tau_{in}, \perp, \beta) \in \mathcal{R}_{T_1}$ for some τ_{in} satisfying the precondition of Γ, and $\beta(\xi) = 1$. Thus, if $\mathcal{R}_T$ is computable for every T, then satisfiability of $[\xi]_{T_1}$ 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) [13, 32], 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 $i \ge 0$, $q_i \in Q$, $\bar{z}_i \in \mathbb{N}^d$, 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 $\bar{z}_m \le \bar{z}_n$?

Both problems are known to be EXPSPACE-complete [36, 43, 32]. In particular, [32] 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 $TS_{ib}(T)$ to $\{0, 1\}$. The vector $\bar{o}$ indicates the current stage of each child T_c of T (init, active or closed) 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 $\bar{x}_{in}^{T_c} \cup \bar{x}_{ret}^{T_c}$ or the value closed. Intuitively, $T_c \notin dom(\bar{o})$ 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 closed, 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 4.1), where $\hat\tau = \tau|(\bar{x}_{in}^T \cup \bar{s}^T)$ and $\hat\tau' = \tau'|(\bar{x}_{in}^T \cup \bar{s}^T)$
- $\bar{c}'_{ib} = \bar{c}_{ib} + \bar{a}$
- if σ' is an internal service, $dom(\bar{o}') = \emptyset$.
- If $\sigma' = \sigma_{T_c}^o$, then $T_c \notin dom(\bar{o})$ and for $\tau_{in}^{T_c} = f_{in}^{-1}(\tau|(\bar{x}_{T_c^\downarrow}^T, h(T_c)))$, for some output $\tau_{out}^{T_c}$ of T_c and truth assignment β^{T_c} to Φ_{T_c}, tuple $(\tau_{in}^{T_c}, \tau_{out}^{T_c}, \beta^{T_c})$ is in $\mathcal{R}_{T_c}$. Note that $\tau_{out}^{T_c}$ can be $\perp$, which indicates that this call to T_c does not return. Also, $\bar{o}' = \bar{o}[T_c \mapsto \tau_{out}^{T_c}]$.
- If $\sigma' = \sigma_{T_c}^c$, then $\bar{o}(T_c) = (f_{out} \circ f_{in}^{-1})(\tau'|(\bar{x}_{T_c^\downarrow}^T \cup \bar{x}_{T_c^\uparrow}^T, h(T_c)))$ and $\bar{o}' = \bar{o}[T_c \mapsto \texttt{closed}]$.
- q' is a successor of q in $B(T, \beta)$ by evaluating Φ_T using (τ', σ'). If $\sigma' = \sigma_{T_c}^o$, 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 $v_0 = (\tau_0, \sigma_0, q_0, \bar{o}_0, \bar{c}_{ib}^0)$ where τ_0 is an initial T-isomorphism type (i.e., for every $x \in \bar{x}_{id}^T - \bar{x}_{in}^T$, $x \sim_{\tau_0}$ null, and for every $x \in \bar{x}_{\mathbb{R}}^T - \bar{x}_{in}^T$, $x \sim_{\tau_0} 0$), $\sigma_0 = \sigma_T^o$, q_0 is the successor of some initial state of $B(T, \beta)$ under (τ_0, σ_0), $dom(\bar{o}_0) = \emptyset$, and $\bar{c}_{ib}^0 = \bar{0}$.

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 (see the extended appendix in [25] for proof).

LEMMA 21. $(\tau_{in}, \tau_{out}, \beta) \in \mathcal{R}_T$ iff there exists an initial state $v_0 = (\tau_0, \sigma_0, q_0, \bar{o}_0, \bar{c}_{ib}^0)$ of $\mathcal{V}(T, \beta)$ for which $\tau_0|\bar{x}_{in}^T = \tau_{in}$ and the following hold:

- If $\tau_{out} \ne \perp$, then there exists state $v_n = (\tau_n, \sigma_n, q_n, \bar{o}_n, \bar{c}_{ib}^n)$ where $\tau_{out} = \tau_n|(\bar{x}_{in}^T \cup \bar{x}_{ret}^T)$, $\sigma_n = \sigma_T^c$, $q_n \in Q^{fin}$ 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 $\tau_{out} = \perp$, then one of the following holds:
  - there exists a state $v_n = (\tau_n, \sigma_n, q_n, \bar{o}_n, \bar{c}_{ib}^n)$ 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 $\bar{z}_n \le \bar{z}'_n$ is called a **lasso path**.
  - There exists state $v_n = (\tau_n, \sigma_n, q_n, \bar{o}_n, \bar{c}_{ib}^n)$ in which $\bar{o}_n(T_c) = \perp$ 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**.

Complexity of Verification We now have all ingredients in place for our verification algorithm. Let Γ be a HAS and $\varphi = [\xi]_{T_1}$ an HLTL-FO formula over Γ. In view of the previous development, $\Gamma \models \varphi$ iff $[\neg\xi]_{T_1}$ is **not** satisfiable by a symbolic tree of runs of Γ. We outline a non-deterministic algorithm for checking satisfiability of $[\neg\xi]_{T_1}$, and establish

	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	$O(\exp(N^c))$	$O(2\text{-}\exp(N^{c \cdot h}))$	$(h+2)\text{-}\exp(O(N))$

Table 1: Space complexity of verification without arithmetic (N: size of (Γ, φ); h: depth of hierarchy; c: constants depending on the schema)

its space complexity $O(f)$, where f is a function of the relevant parameters. The space complexity of verification (the complement) is then $O(f^2)$ by Savitch's theorem [44].

Recall that $[\neg\xi]_{T_1}$ is satisfiable by a symbolic tree of runs of Γ iff $(\tau_{in}, \bot, \beta) \in \mathcal{R}_{T_1}$ for some τ_{in} satisfying the precondition of Γ, and $\beta(\neg\xi) = 1$. By Lemma 21, 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 [32]. 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. 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 C.3 of [25].

THEOREM 22. *Let Γ be a HAS over a fixed schema and φ an HLTL-FO formula over Γ. The deterministic space complexity of checking whether $\Gamma \models \varphi$ is summarized in Table 1.* [4]

Note that the worst-case space complexity is non-elementary, as for feedback-free systems [18]. However, the height of the tower of exponentials in [18] is the square of the total number of artifact variables of the system, whereas in our case it is the depth of the hierarchy, likely to be much smaller.

5. 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 $\mathbb{Q}$), with the same complexity results. 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 4, 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 4, 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 4: 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

---

[4] $k\text{-}\exp$ is the tower of exponential functions of height k.

	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	$O(2\text{-}\exp(N^{c \cdot h}))$	$O(2\text{-}\exp(N^{c \cdot h^2}))$	$(h+2)\text{-}\exp(O(N)))$

Table 2: Space complexity of verification with arithmetic (N: size of (Γ, φ); h: depth of hierarchy; c: constants depending on the schema)

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 [48], 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 [3] from real algebraic geometry 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 Appendix D in the full version of the paper [25].

THEOREM 23. *Let Γ be a HAS over a fixed database schema and φ an HLTL-FO formula over Γ. If arithmetic is allowed in (Γ, φ), then the deterministic space complexity of checking whether $\Gamma \models \varphi$ is summarized in Table 2.*

6. 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 in the model:

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 24. *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 over Γ, whether $\Gamma \models \varphi$.*

The proofs of undecidability for (1)-(7) are by reduction from the Post Correspondence Problem (PCP) [42, 44]. 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, we sketch the proof of undecidability for (2) using this technique in Appendix E of [25].

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 (see [18]). 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 [18], 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, which are satisfied by our example provided in Appendix A.1.

7. RELATED WORK

We have already discussed our own prior related work in the introduction. We summarize next other related work on verification of artifact systems.

Initial work on formal analysis of artifact-based business processes in restricted contexts has investigated reachability [30, 31], general temporal constraints [31], and the existence of complete execution or dead end [11]. 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" [30], restricting to bounded domains [31, 11], or restricting the pre- and post-conditions to be propositional, and thus not referring to data values [31]. [16] adopts an artifact model variation with arithmetic operations but no database. Decidability relies on restricting runs to bounded length. [51] 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. All of these works model no underlying database, sets (artifact relations), task hierarchy, or arithmetic.

A recent line of work has tackled verification of artifact-centric processes with an underlying relational database. [5, 4, 6, 7, 20] evolve the business process model and property language, culminating in [33], which addresses verification of first-order μ-calculus (hence branching time) properties over business processes expressed in a framework that is equivalent to artifact systems whose input is provided by external services. [8, 15] extend the results of [33] 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 [46]. In their unrestricted form, DCDS and HAS have similar expressive power. However, the difference lies in the tackled verification problem and in the restrictions imposed to achieve decidability. We check satisfaction of linear-time properties for every possible choice of initial database instance, whereas the related line checks branching-time properties and assumes that the initial database is given. None of the related works address arithmetic. In the absence of arithmetic, the restrictions introduced for decidability are incomparable (neither subsumes the other).

Beyond artifact systems, there is a plethora of literature on data-centric processes, dealing with various static analysis problems and also with runtime monitoring and synthesis. We discuss the most related works here and refer the reader to the surveys [14, 24] for more. Static analysis for semantic web services is considered in [39], but in a context restricted to finite domains. The works [26, 47, 2] are ancestors of [23] from the context of verification of electronic commerce applications. Their models could conceptually (if not naturally) be encoded in HAS but correspond only to particular cases supporting no arithmetic, sets, or hierarchies. Also, they limit external inputs to essentially come from the active domain of the database, thus ruling out fresh values introduced during the run.

8. CONCLUSION

We showed 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 [18], 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). This brings the verification algorithm closer to practical relevance, particularly since its complexity 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 realistic artifact systems.

9. REFERENCES

[1] Expedia. www.expedia.com. Accessed: 2014-12-10.

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000. Extended abstract in PODS 98.

[3] S. Basu, R. Pollak, and M.-F. Roy. On the number of cells defined by a family of polynomials on a variety. *Mathematika*, 43(1):120–126, 1996.

[4] F. Belardinelli, A. Lomuscio, and F. Patrizi. A computationally-grounded semantics for artifact-centric systems and abstraction results. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 738–743, 2011.

[5] F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of deployed artifact systems via data abstraction. In *Service-Oriented Computing - 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings*, pages 142–156, 2011.

[6] F. Belardinelli, A. Lomuscio, and F. Patrizi. An abstraction technique for the verification of artifact-centric systems. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*, 2012.

[7] F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of GSM-based artifact-centric systems

through finite abstraction. In *Service-Oriented Computing - 10th International Conference, ICSOC 2012, Shanghai, China, November 12-15, 2012. Proceedings*, pages 17–31, 2012.

[8] F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of agent-based artifact systems. *J. Artif. Intell. Res. (JAIR)*, 51:333–376, 2014.

[9] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.

[10] K. Bhattacharya et al. A model-driven approach to industrializing discovery processes in pharmaceutical research. *IBM Systems Journal*, 44(1):145–162, 2005.

[11] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *Proc. Int. Conf. on Business Process Management (BPM)*, pages 288–304, 2007.

[12] BizAgi and Cordys and IBM and Oracle and SAP AG and Singularity (OMG Submitters) and Agile Enterprise Design and Stiftelsen SINTEF and TIBCO and Trisotech (Co-Authors). Case Management Model and Notation (CMMN), FTF Beta 1, Jan. 2013. OMG Document Number dtc/2013-01-01, Object Management Group.

[13] M. Blocklet and S. Schmitz. Model checking coverability graphs of vector addition systems. In *Mathematical Foundations of Computer Science 2011*, pages 108–119. Springer, 2011.

[14] D. Calvanese, G. De Giacomo, and M. Montali. Foundations of data-aware process analysis: a database theory perspective. In *PODS*, pages 1–12, 2013.

[15] D. Calvanese, G. Delzanno, and M. Montali. Verification of relational multiagent systems with data types. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 2031–2037, 2015.

[16] D. Calvanese, G. D. Giacomo, R. Hull, and J. Su. Artifact-centric workflow dominance. In *ICSOC/ServiceWave*, pages 130–143, 2009.

[17] T. Chao et al. Artifact-based transformation of IBM Global Financing: A case study. In *BPM*, 2009.

[18] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM Trans. Database Syst.*, 37(3):22, 2012. Also in ICDT 2011.

[19] E. Damaggio, R. Hull, and R. Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. *Information Systems*, 38:561–584, 2013.

[20] G. De Giacomo, R. D. Masellis, and R. Rosati. Verification of conjunctive artifact-centric services. *Int. J. Cooperative Inf. Syst.*, 21(2):111–140, 2012.

[21] G. De Giacomo and M. 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.

[22] H. de Man. Case management: Cordys approach. BP Trends (`www.bptrends.com`), 2009.

[23] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, pages 252–267, 2009.

[24] A. Deutsch, R. Hull, and V. Vianu. Automatic verification of database-centric systems. *SIGMOD Record*, 43(3):5–17, 2014.

[25] A. Deutsch, Y. Li, and V. Vianu. Verification of hierarchical artifact systems. *arXiv preprint*, arXiv:1604.00967v1, 2016.

[26] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *JCSS*, 73(3):442–474, 2007.

[27] V. Diekert and P. 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.

[28] V. Diekert and P. Gastin. Pure future local temporal logics are expressively complete for Mazurkiewicz traces. *Inf. Comput.*, 204(11):1597–1619, 2006.

[29] E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[30] C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *IEEE International Conference on Service-Oriented Computing and Applications*, 2007.

[31] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *Proceedings of 5th International Conference on Service-Oriented Computing (ICSOC)*, Vienna, Austria, September 2007.

[32] P. 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.

[33] B. B. Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 163–174, 2013.

[34] R. Hull, E. Damaggio, R. D. Masellis, F. Fournier, M. Gupta, F. H. III, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculín. Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events. In *ACM DEBS*, 2011.

[35] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[36] R. Lipton. The reachability problem requires exponential space. *Research Report 62, Department of Computer Science, Yale University, New Haven, Connecticut*, 1976.

[37] M. Marin, R. Hull, and R. Vaculín. Data centric bpm and the emerging case management standard: A short survey. In *BPM Workshops*, 2012.

[38] R. Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297(1):337–354, 2003.

[39] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Intl. World Wide Web Conf. (WWW2002)*, 2002.

[40] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[41] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

[42] E. L. Post. Recursive unsolvability of a problem of Thue. *J. of Symbolic Logic*, 12:1–11, 1947.

[43] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.

[44] M. Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[45] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[46] D. Solomakhin, M. Montali, S. Tessaris, and R. D. Masellis. Verification of artifact-centric systems: Decidability and modeling issues. In *Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*, pages 252–266, 2013.

[47] M. Spielmann. Verification of relational transducers for electronic commerce. *JCSS.*, 66(1):40–65, 2003. Extended abstract in PODS 2000.

[48] A. Tarski. A decision method for elementary algebra and geometry. *1948*, 1951.

[49] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, 1986.

[50] P. Vassiliadis and T. Sellis. A survey of logical models for olap databases. *ACM Sigmod Record*, 28(4):64–69, 1999.

[51] X. Zhao, J. Su, H. Yang, and Z. Qiu. Enforcing constraints on life cycles of business artifacts. In *TASE*, pages 111–118, 2009.

[52] W.-D. Zhu et al. Advanced Case Management with IBM Case Manager. Available at `http://www.redbooks.ibm.com/abstracts/sg247929.html?Open`.

APPENDIX

A. EXAMPLES

In this section we provide an example of HAS modeling a simple travel booking business process similar to Expedia [1]. We also show an example property that the process should satisfy, using HLTL-FO.

A.1 Example Hierarchical Artifact System

The artifact system captures a process where a customer books flights and/or makes hotel reservations. The customer starts with constructing a trip by adding a flight and/or hotel reservation to it. During this time, the customer has the choice to store the trip as a candidate or retrieve a previously stored trip. Once the customer has made a decision, she can proceed to book the trip. If a hotel reservation is made together with certain flights, a discount price may be applied to the hotel reservation. In addition, the hotel reservation can be made by itself, together with the flight, or even after the flight is purchased. After submitting a valid payment, the customer is able to cancel the flight and/or the hotel reservation and receive a refund. If the customer cancels the purchase of a flight, she cannot receive the discount on the hotel reservation.

The Hierarchical artifact system has the following database schema:

- FLIGHTS($\underline{\text{id}}$, price, comp_hotel_id)
  HOTELS($\underline{\text{id}}$, unit_price, 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 dependency FLIGHTS[$comp\_hotel\_id$] $\subseteq$ HOTELS[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 artifact system has 6 tasks: "T1: **ManageTrips**", "T2: **AddHotel**", "T3: **AddFlight**", "T4: **BookInitialTrip**", "T5: **Cancel**" and "T6: **AlsoBookHotel**", which form the hierarchy represented in Figure 1.
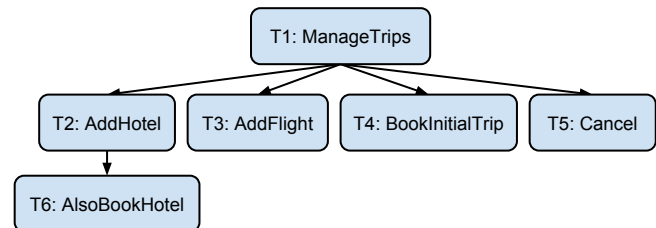

Figure 1: Tasks Hierarchy

The process can be described informally as follows. The customer starts with task **ManageTrips**, where the customer can add a flight and/or hotel to the trip by calling the **AddHotel** or the **AddFlight** tasks. The customer is also allowed to store candidate trips in an artifact relation TRIPS and retrieve previously stored trips. (Note that for simplicity, our example considers only outbound flights in the trip. Return flights can be added by a simple extension to the specification.) After the customer has made a decision, the **BookInitialTrip** task is called to book the trip and the payment is processed. The process also mimics a key feature of Expedia as follows. After payment is made successfully, if the customer booked the flight with no hotel reservation, then she has the opportunity to add a hotel reservation by calling the **AddHotel** task. When she does so, the task **AlsoBookHotel** needs to be called to handle the payment of the added hotel reservation. Note that the **AlsoBookHotel** task can only be called after the flight is booked for but a hotel reservation is missing in the trip. Once the payment is made, the customer can cancel the order by calling the **Cancel** task. Using **Cancel**, the customer is able to cancel the flight and/or the hotel with a full refund. It is important to note that if the customer

cancels the purchase of the flight, then she cannot receive the discount on the hotel reservation.

The tasks are specified below. For convenience, we use existential quantifications in conditions, which can be simulated by adding extra variables. String values are used as syntactic sugar for numeric variables. We assume that the set of strings we used ("Unpaid", "Paid", "FlightCanceled", etc.) correspond to distinct numeric constants. In particular, the string "Unpaid" corresponds to the constant 0. Also for convenience, we use artifact variables with the same names in parent and child tasks. By default, each input/return variable is mapped to the variable in the parent/child task having the same name.

ManageTrips: This is the root task, modeling the process whereby the customer creates, stores, and retrieves candidate trips. A trip consists of a flight and/or hotel reservation. Eventually, one of the candidate trips may be chosen for booking. As the root task, its opening condition is `true` and closing condition is `false`. The task has the following artifact variables:

- ID variables: `flight_id`, `hotel_id`,
- numeric variables: `status` and `amount_paid`

It also has an artifact relation `TRIPS` storing candidate trips (`flight_id`, `hotel_id`). The customer can use the subtasks **AddFlight** and **AddHotel** (specified below) to fill in variables `flight_id` and `hotel_id`. In addition, the task has two internal services: *StoreTrip* and *RetrieveTrip*. Intuitively, when *StoreTrip* is called, the current candidate trip (`flight_id`, `hotel_id`) is inserted into `TRIPS`. When *RetrieveTrip* is called, one tuple is non-deterministically chosen and removed from `TRIPS`, and (`flight_id`, `hotel_id`) is set to be the chosen tuple. The two tasks are specified as follows:

StoreTrip:
Pre-condition: `status` = "Unpaid" ∧ (`flight_id` ≠ `null` ∨ `hotel_id` ≠ `null`)
Post-condition: `flight_id` = `null` ∧ `hotel_id` = `null` ∧ `status` = "Unpaid" ∧ `amount_paid` = 0
Set update: {+`TRIPS`(`flight_id`, `hotel_id`)}

RetrieveTrip:
Pre-condition: `status` = "Unpaid"
Post-condition: `status` = "Unpaid" ∧ `amount_paid` = 0
Set update: {−`TRIPS`(`flight_id`, `hotel_id`)}

AddFlight: This task adds a flight to the trip. It can be opened if `flight_id` = `null` and `status` = "Unpaid" in the parent task. It has no input variable and the return variable is `flight_id`. The task has a single internal service *ChooseFlight* that chooses a flight from the `FLIGHTS` database and stores it in `flight_id`, which is returned to **ManageTrips**.

AddHotel: This task adds a hotel reservation to the trip. It can be opened when `hotel_id` = `null` and `status` is either "Paid" or "Unpaid".

This task has the following artifact variables:

- ID variables: `flight_id`[5], `hotel_id`[6]
- numeric variables: `status`, `amount_paid`, `new_amount_paid` (overwriting `amount_paid` in the parent task when the task returns), `discount_price`, `unit_price` and `hotel_price`

---

[5]the underlined variables are input variables
[6]the wavy underlined variables are return variables

The task has a single internal service *ChooseHotel* which picks a hotel from `HOTELS` and determines the price by checking whether the hotel is compatible with the chosen flight. If they are compatible, then `hotel_price` is set to the discount price, otherwise it is set to the full price.

A hotel can be added to the trip in two scenarios. First, if `status` is "Unpaid", which means that the trip has not been booked, then this task chooses a hotel and the id of the hotel is returned to **ManageTrips**. Second, if `status` is "Paid", which means that a flight has already been purchased without a hotel reservation, then this task chooses a hotel and then the child task **AlsoBookHotel** needs to be called to handle the payment of the newly added hotel. In **AlsoBookHotel**, a payment is received and the new total amount of payment received is written into variable `new_amount_paid` when **AlsoBookHotel** returns.

The closing service of **AddHotel** has condition `status` = "Unpaid" ∨ (`status` = "Paid" ∧ `hotel_price` = `new_amount_paid` − `amount_paid`), which means that either there is no need to call **AlsoBookHotel** or a correct payment has been received in **AlsoBookHotel**. The *ChooseHotel* service is specified as follows:

ChooseHotel:
Pre-condition: `True`
Post-condition:

$$\exists cid \exists p_f \ (\texttt{flight\_id} = \texttt{null} \rightarrow cid = \texttt{null}) \land$$
$$(\texttt{flight\_id} \neq \texttt{null} \rightarrow \texttt{FLIGHTS}(\texttt{flight\_id}, p_f, cid)) \land$$
$$\texttt{HOTELS}(\texttt{hotel\_id}, \texttt{unit\_price}, \texttt{discount\_price}) \land$$
$$(cid = \texttt{hotel\_id} \rightarrow \texttt{hotel\_price} = \texttt{discount\_price}) \land$$
$$(cid \neq \texttt{hotel\_id} \rightarrow \texttt{hotel\_price} = \texttt{unit\_price}) \land$$
$$(\texttt{new\_amount\_paid} = 0)$$

AlsoBookHotel: This task handles payment of hotel reservation made after the flight is purchased. It can be opened if `hotel_id` ≠ `null` and `status` = "Paid" in **AddHotel**. It receives input variables `hotel_price` and `amount_paid` from the parent and has local numeric variables `new_amount_paid` and `hotel_amount_paid`. It has a single service *Pay* which processes the payment. This service simply receives a hotel payment in variable `hotel_amount_paid` and the new total amount of payment received is calculated (`new_amount_paid` = `amount_paid` + `hotel_amount_paid`). The service can fail and the user can retry for unlimited number of times. This task can return only when the payment is successful, which means that the closing condition is `hotel_amount_paid` = `hotel_price`. When **AlsoBookHotel** returns, the numeric variable `new_amount_paid` is returned to **ManageTrips**.

BookInitialTrip: This task allows the customer to reserve and pay for the chosen trip. Its opening condition is `status` = "Unpaid". This task has the following variables:

- ID variables: `flight_id`, `hotel_id`
- numeric variables: `status`, `amount_paid`, `ticket_price`, `hotel_price`

The task contains a single service *Pay* to process the payment, which can fail and be retried for an unlimited number of times. Note that if the trip contains both the flight and hotel, when *Pay* is called, the payments for both of them are received.

If the payment is successful (i.e. `amount_paid` equals to the flight price plus the hotel price), `status` is set to "Paid".

Otherwise it is set to "Failed". The closing condition of this task is status = "Paid" or status = "Failed". When **BookInitialTrip** returns, status and amount_paid in the parent task are updated by the new status and amount_paid returned by **BookInitialTrip**. The *Pay* service is specified as follows:

Pay:
Pre-condition: $\texttt{hotel\_id} \neq \texttt{null} \vee \texttt{flight\_id} \neq \texttt{null}$
Post-condition:

$\exists cid \exists p_1 \exists p_2$
$(\texttt{flight\_id} = \texttt{null} \rightarrow \texttt{ticket\_price} = 0 \wedge cid = \texttt{null}) \wedge$
$(\texttt{flight\_id} \neq \texttt{null} \rightarrow \texttt{FLIGHTS}(\texttt{flight\_id}, \texttt{ticket\_price},$
$cid)) \wedge (\texttt{hotel\_id} = \texttt{null} \rightarrow \texttt{hotel\_price} = 0) \wedge$
$(\texttt{hotel\_id} \neq \texttt{null} \rightarrow (\texttt{HOTELS}(\texttt{hotel\_id}, p_1, p_2) \wedge$
$(\texttt{hotel\_id} = cid \rightarrow \texttt{hotel\_price} = p_2) \wedge$
$(\texttt{hotel\_id} \neq cid \rightarrow \texttt{hotel\_price} = p_1)) \wedge$
$(\texttt{amount\_paid} = \texttt{ticket\_price} + \texttt{hotel\_price} \rightarrow$
$\texttt{status} = \text{"Paid"}) \wedge (\texttt{amount\_paid} \neq \texttt{ticket\_price} +$
$\texttt{hotel\_price} \rightarrow \texttt{status} = \text{"Failed"})$

Cancel: In this task, the customer can cancel the flight and/or hotel after the trip has been paid for. Its opening condition is status = "Paid". This task has the following variables:

- ID variables: <u>hotel_id</u> and <u>flight_id</u>
- numeric variables: <u>amount_paid</u>, ticket_price, discount_price, unit_price, hotel_price, amount_refunded and <u>status</u>

The task has 3 services, *CancelFlight*, *CancelHotel* and *CancelBoth* which cancel the flight, the hotel reservation, or both of them, respectively. When any of these services is called, amount_refunded is calculated to be the correct amount needs to be refunded to the customer and status is set to "FlightCanceled", "HotelCanceled" and "AllCanceled" respectively. In particular, if the customer would like to cancel the flight while keeping the hotel reservation, and if a discount has been applied on the hotel reservation, then the correct amount_refunded equals to ticket_price minus the difference between the normal cost and the discounted cost of the hotel since she is no longer eligible for the discount.

The closing condition of this task is True. We show the specification of *CancelFlight* as an example. Let Discounted be the subformula

$(\texttt{hotel\_id} \neq \texttt{null}) \wedge (\texttt{hotel\_price} = \texttt{discount\_price})$

And let Penalized be the subformula

$\texttt{amount\_refunded} = \texttt{ticket\_price} -$
$(\texttt{unit\_price} - \texttt{discount\_price})$

CancelFlight:
Pre-condition:

$\texttt{flight\_id} \neq \texttt{null} \wedge \texttt{status} \neq \text{"FlightCanceled"} \wedge$
$\texttt{status} \neq \text{"HotelCanceled"} \wedge \texttt{status} \neq \text{"AllCanceled"}$

Post-condition:

$\exists cid \; \texttt{FLIGHTS}(\texttt{flight\_id}, \texttt{ticket\_price}, cid) \wedge$
$(\texttt{hotel\_price} = \texttt{amount\_paid} - \texttt{ticket\_price}) \wedge$
$(\texttt{hotel\_id} \neq \texttt{null} \rightarrow$
$(\texttt{HOTELS}(\texttt{hotel\_id}, \texttt{unit\_price}, \texttt{discount\_price}) \wedge$
$(\neg \texttt{Discounted} \rightarrow \texttt{amount\_refunded} = \texttt{ticket\_price}) \wedge$
$(\texttt{Discounted} \rightarrow \texttt{Penalized}) \wedge \texttt{status} = \text{"FlightCanceled"}$

A.2 Example HLTL-FO Property

Suppose we wish to enforce the following policy: *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 pay for the flight, then reserve the hotel with the discount price, but next cancel the flight without penalty. Detecting such bugs can be subtle, especially in a system allowing concurrency. The following HLTL-FO property of task **ManageTrips** says "If *AddHotel* is called and a hotel reservation is added with a discounted price, then at the task *Cancel*, if the customer would like to cancel the flight, a penalty must be paid".

The property is specified as $[\varphi]_{\texttt{T1}}$ where φ is the formula:

$\varphi = \mathbf{F}[\mathbf{F} \; (\texttt{Discounted} \wedge \mathbf{X} \; \sigma^o_{\texttt{T6:AlsoBookHotel}})]_{\texttt{T2:AddHotel}} \rightarrow$
$\mathbf{G}(\sigma^o_{\texttt{T5:Cancel}} \rightarrow [\mathbf{G}(\textit{CancelFlight} \rightarrow \texttt{Penalized})]_{\texttt{T5:Cancel}})$

with the subformulas Discounted and Penalized defined above.

Notice that in the specification there is no guard preventing *AddHotel* and *Cancel* to run concurrently after a successful payment is made, which can lead to a violation of this property. The problem can be fixed by adding a new variable in **ManageTrips** to indicate whether **AddHotel** or **Cancel** are currently running and modifying their opening conditions to make sure that these two tasks are mutual exclusive.