VERIFAS: A Practical Verifier for Artifact Systems

Yuliang Li
UC San Diego
La Jolla, California
yul206@eng.ucsd.edu

Alin Deutsch
UC San Diego
La Jolla, California
deutsch@cs.ucsd.edu

Victor Vianu
UC San Diego & INRIA Paris
La Jolla, California
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 research bridges the gap between the theory and practice of artifact verification with VERIFAS, the first implementation of practical significance of an artifact verifier with full support for unbounded data. VERIFAS verifies within seconds linear-time temporal properties over real-world and synthetic workflows of complexity in the range recommended by software engineering practice. Compared to our previous implementation based on the widely-used Spin model checker, VERIFAS not only supports a model with richer data manipulations but also outperforms it by over an order of magnitude. VERIFAS' good performance is due to a novel symbolic representation approach and a family of specialized optimizations.

PVLDB Reference Format:
Yuliang Li, Alin Deutsch, and Victor Vianu. VERIFAS: A Practical Verifier for Artifact Systems. *PVLDB*, 11(3): 283 - 296, 2017.
DOI: 10.14778/3157794.3157798

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 [35], successfully deployed in practice [7, 6, 10, 14, 49] and adopted in industrial standards.

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) [12, 28]. 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) [34, 36].

Proceedings of the VLDB Endowment, Vol. 11, No. 3

DOI: 10.14778/3157794.3157798

Artifact systems deployed in industrial settings typically specify complex workflows prone to costly bugs, whence the need for verification of critical properties. Over the past few years, the verification problem for artifact systems has been intensively studied. Rather than relying on general-purpose software verification tools suffering from well-known limitations, the focus of the research community has been 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. However, verification was shown to be decidable for significant classes of properties and artifact models.

The present paper bridges the gap between the theory and practice of artifact verification by studying the implementation of a full-fledged and efficient artifact verifier. The artifact model we verify is a variant of the Hierarchical Artifact System (HAS) model presented in [17]. In brief, a HAS consists of a database and a hierarchy (rooted tree) of *tasks*. Each task has associated to it local evolving data consisting of a tuple of artifact variables and an updatable artifact relation. It also has an associated set of *services*. Each application of a service is guarded by a pre-condition on the database and local data and causes an update of the local data, specified by a post condition (constraining the next artifact tuple) and an insertion or retrieval of a tuple from the artifact relation. In addition, a task may invoke a child task with a tuple of parameters, and receive back a result if the child task completes. A run of the artifact system is obtained by any valid interleaving of concurrently running task services. Properties of HAS are specified using an extension of Linear-Time Temporal logic (LTL).

In a previous study [32], we made a first attempt at implementing a verifier for a simple version of HAS using Spin [27], the verification tool widely used in the model checking community. However, as discussed in [32], Spin cannot handle some of the most useful features of artifacts which support unbounded data, such as *sets* of tuples (see Section 2 for details). Moreover, its performance is disappointing even after deploying a battery of non-trivial optimizations. This indicates the limited applicability of existing tools for HAS verification and suggests the need for tailored approaches.

In this paper we present VERIFAS, an artifact verifier implementation built from scratch. Our main contributions are the following.

- We define HAS*, a novel variant of HAS which strikes a more practically relevant trade-off between expressivity and verification complexity, as demonstrated by its ability to specify a realistic set of business processes. We adapt to HAS* the theory developed in [17], laying the groundwork for our implementation.

- We implement VERIFAS, a fully automatic verifier for HAS*. The implementation makes crucial use of *novel optimization techniques*, with dramatic impact on performance. The optimizations are non-trivial and include concise symbolic representations, aggressive pruning in the search algorithm, and the use of highly efficient data structures.
- We evaluate the performance of VERIFAS using both real-world and synthetic artifact systems and properties from a benchmark we create, bootstrapping from existing sets of business process specifications and properties by extending them with data-aware features. To our knowledge, this is the first benchmark for business processes and properties that includes such data aware features. The experiments highlight the impact of the various optimizations and parameters of both the artifact systems and properties.
- We adapt to HAS* a standard complexity measure of control flow used in software engineering, *cyclomatic complexity* [48], and show experimentally, using the above benchmark, that cyclomatic complexity of HAS* specifications correlates meaningfully with verification times. Since conventional wisdom in software engineering holds that well-designed, human readable programs have relatively low cyclomatic complexity, this is an indication that verification times are likely to be good for well-designed HAS* specifications.

Taking this and other factors into account, the experimental results show that our verifier performs very well on practically relevant classes of artifact systems. Compared to the Spin-based verifier of [32], it not only applies to a much broader class of artifacts but also has a decisive performance advantage even on the simple artifacts the Spin-based verifier is able to handle. To the best of our knowledge, this is the first implementation of practical significance of an artifact verifier with full support for unbounded data.

The paper is organized as follows. We start by introducing in Section 2 the HAS* model supported by VERIFAS, and we review LTL-FO, the temporal logic for specifying properties of HAS*. Section 3 describes the implementation of VERIFAS by first reviewing in brief the theory developed of [17], particularly the symbolic representation technique used in establishing the theoretical results. We show an extension of the symbolic representation, called partial isomorphism type, to allow practical verification by adapting the classic Karp-Miller algorithm [29]. We then introduce three specialized optimizations to gain further performance improvement. We present our experimental results in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2. THE MODEL

In this section we present the variant of Hierarchical Artifact Systems used in our study. The variant, denoted HAS*, differs from the HAS model used in [17] in two respects. On one hand, it *restricts* HAS as follows:

- it disallows arithmetic in service pre-and-post conditions
- the underlying database schema uses an *acyclic* set of foreign keys

On the other hand, HAS* *extends* HAS by removing various restrictions:

- tasks may have multiple updatable artifact relations
- each subtask of a given task may be called multiple times between task transitions
- certain restrictions on how variables are passed as parameters among tasks, or inserted/retrieved from artifact relations, are lifted

Because HAS* imposes some restrictions on HAS but removes others, it is incomparable to HAS. Intuitively, the choice of HAS* over HAS as a target for verification is motivated by the fact that HAS* achieves a more appealing trade-off between expressiveness and verification complexity. The acyclic schema restriction, satisfied by the widely used Star (or Snowflake) schemas [30, 47], is acceptable in return for the removal of various HAS restrictions limiting modeling capability. Indeed, as shown by our real-life examples, HAS* is powerful enough to model a wide variety of business processes. While the current version of VERIFAS does not handle arithmetic, the core verification algorithm can be augmented to include arithmetic along the lines developed for HAS in [17]. Limited use of aggregate functions can also be accommodated. These enhancements are left for future work.

We now present the syntax and semantics of HAS*. The formal definitions below are illustrated with an intuitive example of the HAS* specification of a real-world order fulfillment business process originally written in BPMN [2]. The workflow allows customers to place orders and the supplier company to process the orders. A detailed description of the example can be found in the appendix of [18].

We begin by defining 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 $\{\text{ID}, F_1, \ldots, F_m\}$.*

To each foreign key attribute F_i of R is associated a relation R_{F_i} of $\mathcal{DB}$ and the inclusion dependency[1] *$R[F_i] \subseteq R_{F_i}[\text{ID}]$. It is said that F_i references R_{F_i}.*

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

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. All database schemas considered in this paper are acyclic.

EXAMPLE 2. *The order fulfillment workflow has the following database schema:*

- `CUSTOMERS`(ID, `name`, `address`, `record`)
  `ITEMS`(ID, `item_name`, `price`)
  `CREDIT_RECORD`(ID, `status`)

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

We assume two infinite, disjoint domains of IDs and data values, denoted by DOM_{id} and DOM_{val}, and an additional constant `null` where `null` $\notin DOM_{id} \cup DOM_{val}$ (`null` serves as a convenient default initialization value). The domain of all non-key attributes is DOM_{val}. The domain of each key attribute ID of relation R is an infinite subset $Dom(R.ID)$ of DOM_{id}, and $Dom(R.ID) \cap$

[1] The inclusion uses set semantics.

$Dom(R'.ID) = \emptyset$ for $R \neq R'$. The domain of a foreign key attribute F referencing R is $Dom(R.ID)$. 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 in foreign keys referencing it. An *instance* of a database schema $\mathcal{DB}$ is a mapping D associating to each relation symbol R a finite relation (set of tuples) $D(R)$ 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 $\texttt{adom}(D)$, consists of all elements of D.

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

Definition 3. *A **task schema** over database schema $\mathcal{DB}$ is a tuple $T = \langle \bar{x}^T, \mathcal{S}^T, \bar{x}^T_{in}, \bar{x}^T_{out} \rangle$ where $\bar{x}^T$ is a sequence of artifact variables, $\mathcal{S}^T$ is a set of relation symbols not in $\mathcal{DB}$, and $\bar{x}^T_{in}$ and $\bar{x}^T_{out}$ are subsequences of $\bar{x}^T$. For each relation $\mathcal{S} \in \mathcal{S}^T$, we denote by $\texttt{attr}(\mathcal{S})$ the set of attributes of $\mathcal{S}$. The domain of each variable $x \in \bar{x}^T$ and each attribute $A \in \texttt{attr}(\mathcal{S})$ is either $\text{DOM}_{val} \cup \{\texttt{null}\}$ or $dom(R.ID) \cup \{\texttt{null}\}$ for some relation $R \in \mathcal{DB}$. In the latter case we say that the type of x (or A) is $\texttt{type}(x) = R.ID$ ($\texttt{type}(A) = R.ID$). An **instance** ρ of T is a tuple (ν, S) where ν is a valuation of $\bar{x}^T$ and S is an instance of $\mathcal{S}^T$ such that $S(\mathcal{S})$ is of the type of $\mathcal{S}$ for each $\mathcal{S} \in \mathcal{S}^T$.*

We refer to the relations in $\mathcal{S}^T$ as the *artifact relations* of T and to $\bar{x}^T_{in}$ and $\bar{x}^T_{out}$ as the input and output variables of T. We denote by $\bar{x}^T_{id} = \bar{x}^T \cap VAR_{id}$ and $\bar{x}^T_{val} = \bar{x}^T \cap VAR_{val}$.

Example 4. *The order fulfillment workflow has a task called **ProcessOrders**, which stores the order data and processes the orders by interacting with other tasks. It has the following artifact variables:*

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

There are no input or output variables. The task also has an artifact relation $\texttt{ORDERS}(\texttt{cust\_id}, \texttt{item\_id}, \texttt{status}, \texttt{instock})$ *with attributes of the same types as the variables. Intuitively,* `ORDERS` *stores the orders to be processed, where each order consists of a customer and an ordered item. The variable* `status` *indicates the current status of the order and* `instock` *indicates whether the item is currently in stock.*

We next define artifact schemas, essentially a hierarchy of task schemas with an underlying database.

Definition 5. *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 $\mathcal{S}_{\mathcal{H}}$ the relational schema $\cup_{1 \leq i \leq k} \mathcal{S}^{T_i}$. An instance of $\mathcal{S}_{\mathcal{H}}$ is a mapping associating to each $S \in \mathcal{S}_{\mathcal{H}}$ a finite relation of the same type.

Example 6. *The order fulfillment workflow has 5 tasks: T_1: **ProcessOrders**, T_2:**TakeOrder**, T_3:**CheckCredit**, T_4: **Restock** and T_5:**ShipItem**, which form the hierarchy represented in Figure 1. Intuitively, the root task **ProcessOrders** serves as a global coordinator which maintains a pool of all orders and the child tasks **TakeOrder**, **CheckCredit**, **Restock** and **ShipItem** implement the 4 sequential stages in the fulfillment of an order. At a high level, **ProcessOrders** repeatedly picks an order from its pool and processes it with a stage by calling the corresponding child task. After the child task returns, the order is either placed back into the pool or processed with the next stage. For each order, the workflow first obtains the customer and item information using the **TakeOrder** task. The credit record of the customer is checked by the **CheckCredit** task. If the record is good, then **ShipItem** can be called to ship the item to the customer. If the requested item is unavailable, then **Restock** must be called before **ShipItem** to procure the item.*

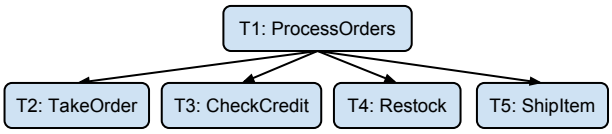

Figure 1: Tasks Hierarchy

Definition 7. *An **instance** of an artifact schema $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$ is a tuple $I = \langle \nu, stg, D, S \rangle$ where D is a finite instance of $\mathcal{DB}$, S a finite instance of $\mathcal{S}_{\mathcal{H}}$, ν a valuation of $\bigcup_{i=1}^{k} \bar{x}^{T_i}$, and stg (standing for "stage") a mapping of $\{T_1, \ldots, T_k\}$ to $\{\texttt{active}, \texttt{inactive}\}$.*

The stage $stg(T_i)$ of a task T_i has the following intuitive meaning in the context of a run of its parent: `active` says that T_i has been called and has not yet returned its answer, and `inactive` indicates that T_i is not active. A task T_i can be called any number of times within a given run of its parent, but only one instance of it can be active at any given time.

Example 8. *Figure 2 shows a partial example of an instance of the Order Fulfillment artifact system. The only active task is **ProcessOrder**.*

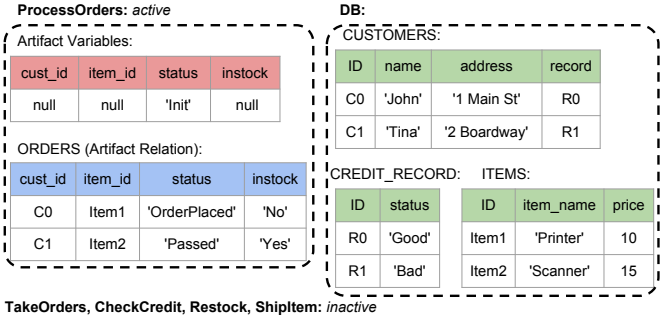

Figure 2: An instance of the Order Fulfillment workflow

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 $\mathcal{DB} \cup \{=\}$ whose variables are included in $\bar{x}$. The special constant `null` can be used in equalities. 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 $\{y_1, \ldots, y_m\} \subseteq VAR_{val}$. If α is a

condition on $\bar{x}$, D an instance of $\mathcal{DB}$ and ν a valuation of $\bar{x}$, we denote by $D \models \alpha(\nu)$ the fact that D satisfies α with valuation ν, with standard semantics. For an atom $R(\bar{y})$ in α where $R \in \mathcal{DB}$, if $\nu(y) = \texttt{null}$ for some $y \in \bar{y}$, then $R(\nu(\bar{y}))$ is false (because null does not occur in database relations). Although conditions are quantifier-free, $\exists$FO conditions can be easily simulated by adding variables to $\bar{x}^T$, so we use them as shorthand whenever convenient.

EXAMPLE 9. *The $\exists$FO condition $\exists n \exists a \exists r$ CUSTOMERS(cust_id, n, a, r)$\wedge$CREDIT_RECORD(r, "Good") states that the customer with ID* cust_id *has good credit.*

We next define services of tasks. We start with internal services, which update the artifact variables and artifact relation of the task. Intuitively, internal services implement local actions, that do not involve any other task.

DEFINITION 10. *Let $T = \langle \bar{x}^T, \mathcal{S}^T, \bar{x}^T_{in}, \bar{x}^T_{out} \rangle$ be a task of an artifact schema $\mathcal{A}$. An **internal service** σ of T is a tuple $\langle \pi, \psi, \bar{y}, \delta \rangle$ where:*

- *π and ψ, called* pre-condition *and* post-condition*, respectively, are conditions over $\bar{x}^T$*
- *$\bar{y}$ is the set of* propagated variables*, where $\bar{x}^T_{in} \subseteq \bar{y} \subseteq \bar{x}^T$;*
- *δ, called the* update*, is a subset of $\{+\mathcal{S}_i(\bar{z}), -\mathcal{S}_i(\bar{z}) | \mathcal{S}_i \in \mathcal{S}^T, \bar{z} \subseteq \bar{x}^T, \texttt{type}(\bar{z}) = \texttt{type}(\texttt{attr}(\mathcal{S}_i))\}$ of size at most 1.*
- *if $\delta \neq \emptyset$ then $\bar{y} = \bar{x}^T_{in}$*

Intuitively, an internal service σ of T can be called only when the current instance satisfies the pre-condition π. The update on variables $\bar{x}^T$ is valid if the next instance satisfies the post-condition ψ and the values of propagate variables $\bar{y}$ stay unchanged.

Any task variable that is not propagated can be changed arbitrarily during a task activation, as long as the post condition holds. This feature allows services to also model actions by external actors who provide input into the workflow by setting the value of non-propagated variables. Such actors may even include humans or other parties whose behavior is not deterministic. For example, a bank manager carrying out a "loan decision" action can be modeled by a service whose result is stored in a non-propagated variable and whose value is restricted by the post-condition to either "Approve" or "Deny". Note that deterministic actors are modeled by simply using tighter post-conditions.

When $\delta = \{+\mathcal{S}_i(\bar{z})\}$, a tuple containing the *current* value of $\bar{z}$ is inserted into $\mathcal{S}_i$. When $\delta = \{-\mathcal{S}_i(\bar{z})\}$, a tuple is chosen and removed from $\mathcal{S}_i$ and the *next* value of $\bar{z}$ is assigned with the value of the tuple. Note that $\bar{x}^T_{in}$ are always propagated, and no other variables are propagated if $\delta \neq \emptyset$. The restriction on updates and variable propagation may at first appear mysterious. Its underlying motivation is that allowing simultaneous artifact relation updates and variable propagation turns out to raise difficulties for verification, while the real examples we have encountered do not require this capability.

EXAMPLE 11. *The **ProcessOrders** task has 3 internal services:* Initialize*,* StoreOrder *and* RetrieveOrder*. Intuitively,* Initialize *creates a new order with* cust_id $=$ item_id $=$ null*. When* RetrieveOrder *is called, an order is chosen non-deterministically and removed from* ORDERS *for processing, and* (cust_id, item_id, status, instock) *is set to be the chosen tuple. When* StoreOrder *is called, the current order* (cust_id, item_id, status, instock) *is inserted into* ORDERS*. The latter two services are specified as follows.*

RetrieveOrder*:*
Pre: cust_id $=$ null $\wedge$ item_id $=$ null
Post: True
Update: $\{-$ORDERS(cust_id, item_id, status, instock)$\}$

StoreOrder*:*
Pre: cust_id $\neq$ null $\wedge$ item_id $\neq$ null $\wedge$ status $\neq$ *"Failed"*
Post: cust_id $=$ null $\wedge$ item_id $=$ null $\wedge$ status $=$ *"Init"*
Update: $\{+$ORDERS(cust_id, item_id, status, instock)$\}$
The sets of propagated variables are empty for both services.

An internal service of a task T specifies transitions that modify the variables $\bar{x}^T$ of T and the contents of $\mathcal{S}^T$. Figure 3 shows an example of a transition that results from applying the service *StoreOrder* of the **ProcessOrders** task.

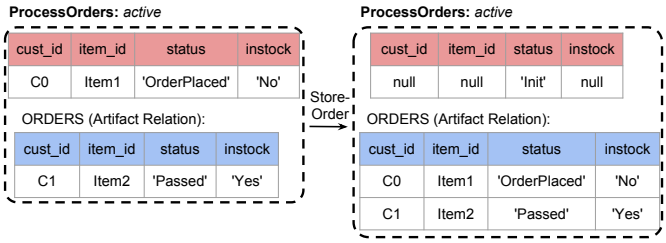

Figure 3: Transition caused by an internal service

As seen above, internal services of a task cause transitions on the data local to the task. Interactions among tasks are specified using two kinds of special services, called the *opening-services* and *closing-services*. Specifically, each task T is equipped with an opening service σ^o_T and a closing service σ^c_T. Each non-root task T can be activated by its parent task via a call to σ^o_T which includes passing parameters to T that initialize its input variables $\bar{x}^T_{in}$. When T terminates (if ever), it returns to the parent the contents of its output variables $\bar{x}^T_{out}$ via a call to σ^c_T. Moreover, calls to σ^o_T are guarded by a condition on the parent's artifact variables, and closing calls to σ^c_T are guarded by a condition on the artifact variables of T. The formal definition is provided in the extended version [18].

For uniformity of notation, we also equip the root task T_1 with a service $\sigma^o_{T_1}$ with pre-condition *true* 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, $\Sigma^{oc}_T = \Sigma_T \cup \{\sigma^o_T, \sigma^c_T\}$, and $\Sigma^{obs}_T = \Sigma^{oc}_T \cup \{\sigma^o_{T_c}, \sigma^c_{T_c} \mid T_c \in child(T)\}$. Intuitively, Σ^{obs}_T consists of the services observable locally in runs of task T.

EXAMPLE 12. *As the root task, the opening condition of **ProcessOrders** is* True *and closing condition is* False*. All variables are initialized to* null*.*

*The opening condition of **TakeOrder** is* status $=$ *"Init" in task **ProcessOrders**, meaning that the customer and item information have not yet been entered by the customer. The task contains* cust_id*,* item_id*,* status *and* instock *as variables (with no input variable). When this task is called, the customer enters the information of the order (*cust_id *and* item_id*) and the status of the order is set to "OrderPlaced". An external service determines whether the item is in stock or not and sets the value of* instock *accordingly. All variables are output variables returned to the parent task. The closing condition is* cust_id $\neq$ null $\wedge$ item_id $\neq$ null*. When it holds, **TakeOrder** can be closed, and the values of these variables are passed to **ProcessOrders** (to the variables with the same names[2]). Figure 4 illustrates a transition caused by the closing service of **TakeOrder**.*

We are finally ready to define HAS*.

DEFINITION 13. *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*

[2]While the formal definition disallows using the same variable names in different tasks, we do so for convenience, since the variable names can be easily disambiguated using the task name.

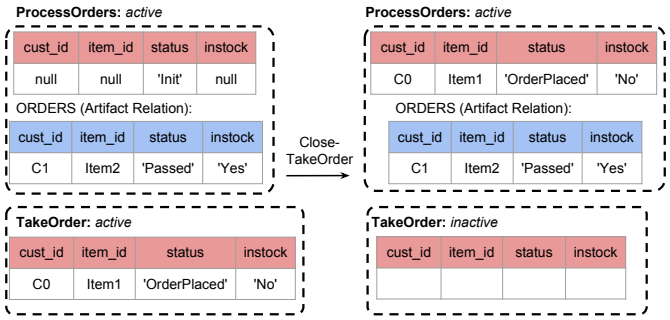

Figure 4: Transition with a Closing Service

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 $\bar{x}^{T_1}$ (the global pre-condition of Γ), where T_1 is the root task.

We next define the semantics of HAS*. Intuitively, a run of a HAS* on a database D consists of an infinite sequence of transitions among HAS* instances (also referred to as configurations, or snapshots), starting from an initial artifact tuple satisfying pre-condition Π, and empty artifact relations. The intuition is that at each snapshot, a transition can be made at an active task T by applying either an internal service of T, the opening service of an inactive subtask T_c, or the closing service of T. In addition, we require that an internal service of T can only be applied after all active subtasks of T have returned their answer. Given two instances I, I' and a service σ, we denote by $I \xrightarrow{\sigma} I'$ if there is a valid transition from I to I' by applying σ. The full definition of transitions can be found in the appendix of [18].

We next define *runs* of artifact systems. We will assume that runs are *fair*, i.e. no task is starved forever by other running tasks. Fairness is commonly ensured by schedulers in multi-process systems. We also assume that runs are non-blocking, i.e. for each task that has not yet returned its answer, there is a service applicable to it or to one of its descendants.

DEFINITION 14. *Let $\Gamma = \langle \mathcal{A}, \Sigma, \Pi \rangle$ be an artifact system, where $\mathcal{A} = \langle \mathcal{H}, \mathcal{DB} \rangle$. A* 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 $\mathcal{A}$, $\sigma_i \in \Sigma$, $\sigma_0 = \sigma^o_{T_1}$, $D \models \Pi(\nu_0)$, $stg_0 = \{T_1 \mapsto \texttt{active}, T_i \mapsto \texttt{inactive} \mid 2 \leq i \leq k\}$, $S_0 = \{\mathcal{S}_\mathcal{H} \mapsto \emptyset\}$, and for each $i > 0$ $I_{i-1} \xrightarrow{\sigma_i} I_i$. In addition, for each $i \geq 0$ and task T active in I_i, there exists $j > i$ such that $\sigma_j \in \bigcup_{T' \in \text{desc}^*(T)} \Sigma^{oc}_{T'}$.*

We denote by $Runs(\Gamma)$ the set of runs of Γ. Observe that all runs of Γ are infinite. In a given run, the root task itself may have an infinite run, or other tasks may have infinite runs. However, if a task T has an infinite run, then none of its ancestor tasks can make an internal transition or return (although they can still call other children tasks).

Because of the hierarchical structure of HAS*, and the locality of task specifications, the actions of independent tasks running concurrently can be arbitrarily interleaved. In order to express properties of HAS* in an intuitive manner, it will be useful to ignore such interleavings and focus on the *local runs* of each task, consisting of the transitions affecting the local variables and artifact relations of the task, as well as interactions with its children tasks. A local run of T induced by ρ is a subsequence ρ_T of ρ corresponding to transitions caused by T's observable services (call these *observable T-transitions*). ρ_T starts from an opening service T-transition and includes all subsequent observable T-transitions up to the first occurrence of a closing service T-transition (if any). See the appendix of [18] for the formal definition. We denote by $Runs_T(\rho)$ the set of local runs of T induced by the run ρ of Γ, and $Runs_T(\Gamma) = \bigcup_{\rho \in Runs(\Gamma)} Runs_T(\rho)$.

2.1 Specifying properties of artifact systems

In this paper we focus on verifying temporal properties of local runs of tasks in an artifact system. For instance, in a task implementing an e-commerce application, we would like to specify properties such as:

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

In order to specify such temporal properties we use, as in previous work, an extension of LTL (linear-time temporal logic). LTL is propositional logic augmented with temporal operators such as **G** (always), **F** (eventually), **X** (next) and **U** (until) (e.g., see [37]). An LTL formula φ with propositions $prop(\varphi)$ defines a property of sequences of truth assignments to $prop(\varphi)$. For example, $\mathbf{G}p$ says that p always holds in the sequence, $\mathbf{F}p$ says that p will eventually hold, $p\mathbf{U}q$ says that p holds at least until q holds, and $\mathbf{G}(p \rightarrow \mathbf{F}q)$ says that whenever p holds, q must hold subsequently.

An LTL-FO property[3] of a task T is obtained starting from an LTL formula using some set $P \cup \Sigma^{obs}_T$ of propositions. Propositions in P are interpreted as conditions over the variables $\bar{x}^T$ of T together with some additional *global* variables $\bar{y}$, shared by different conditions and allowing to connect the states of the task at different moments in time. The global variables are universally quantified over the entire property. Recall that Σ^{obs}_T consists of the services observable in local runs of T (including calls and returns from children tasks). A proposition $\sigma \in \Sigma^{obs}_T$ indicates the application of service σ in a given transition.

LTL-FO is formally defined in the appendix of [18]. We provide a flavor thereof using the example property (†). The property is of the form $\varphi = \mathbf{G}(p \rightarrow (\neg q \ \mathbf{U} \ r))$, which means if p happens, then in what follows, q will not happen until r is true. Here p says that the **TakeOrder** task returned with an out-of-stock item, q states that the **ShipItem** task is called with the same item, and r states that the service **Restock** is called to restock the item. Since the item mentioned in p, q and r must be the same, the formula requires using a global variable i to record the item ID. This yields the following LTL-FO property:

$$\forall i \ \mathbf{G}((\sigma^c_{\texttt{TakeOrder}} \wedge \texttt{item\_id} = i \wedge \texttt{instock} = \text{“No”}) \rightarrow (\neg(\sigma^o_{\texttt{ShipItem}} \wedge \texttt{item\_id} = i) \ \mathbf{U} \ (\sigma^o_{\texttt{Restock}} \wedge \texttt{item\_id} = i)))$$

A correct specification can enforce (†) simply by requiring in the pre-condition of $\sigma^o_{\texttt{ShipItem}}$ that the item is in stock. One such pre-condition is ($\texttt{instock} = \text{“Yes”} \wedge \texttt{status} = \text{“Passed”}$), meaning that the item is in stock and the customer passed the credit check. However, in a similar specification where the $\texttt{instock} = \text{“Yes”}$ test is performed within **ShipItem** (i.e. in the pre-conditions of all shipping internal services) instead of the opening service of **ShipItem**, the LTL-FO property (†) is violated because **ShipItem** can be opened without first calling the **Restock** task. Our verifier would detect this error and produce a counter-example illustrating the violation.

We say that a local run ρ_T of task T satisfies $\forall \bar{y} \varphi_f$, where $prop(\varphi) = P \cup \Sigma^{obs}_T$, if φ is satisfied, for all valuations of $\bar{y}$ in $DOM_{id} \cup DOM_{val} \cup \{\texttt{null}\}$, by the sequence of truth assignments to $P \cup \Sigma^{obs}_T$ induced by f on ρ_T. More precisely, let (I_i, σ_i) denote the i^{th} snapshot of ρ_T. For each $p \in P$, the truth value induced for p in

[3]The variant of LTL-FO used here differs from some previously defined in that the FO formulas interpreting propositions are quantifier-free. By slight abuse we use here the same name.

(I_i, σ_i) is the truth value of the condition $f(p)$ in I_i; a proposition $\sigma \in \Sigma_T^{obs}$ holds in (I_i, σ_i) if $\sigma_i = \sigma$. A task T satisfies $\forall \bar{y} \varphi_f$ if ρ_T satisfies $\forall \bar{y} \varphi_f$ for every $\rho_T \in \mathit{Runs}_T(\Gamma)$. Note that the database is fixed for each run, but may be different for different runs.

A classical result in model checking states that for every LTL formula φ, one can construct a finite-state automaton B_φ, called a Büchi automaton, that accepts precisely the infinite sequences of truth assignments to $prop(\varphi)$ that satisfy φ. A Büchi automaton is syntactically just a finite-state automaton, which accepts an infinite word if it goes infinitely often through an accepting state [46, 43]. Here we are interested in evaluating LTL-FO formulas $\forall \bar{y} \varphi_f$ on both infinite *and* finite runs (infinite runs occur when a task runs forever). 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 classical finite-state automaton with final states Q^{fin} accepts precisely the finite words that satisfy φ.

REMARK 15. *In [17] we consider a more complex logic for specifying properties of artifact systems, called Hierarchical LTL-FO (HLTL-FO). Intuitively, an HLTL-FO formula uses as building blocks LTL-FO formulas as above, acting on local runs of individual tasks, but can additionally recursively state HLTL-FO properties on runs resulting from calls to children tasks. As shown in [17], verification of HLTL-FO properties can be reduced to satisfiability of LTL-FO properties by individual tasks. Our implementation focuses on verification of LTL-FO properties of individual tasks. While this could be used as a building block for verifying complex HLTL-FO properties, verification of LTL-FO properties of individual tasks is in fact adequate in most practical situations we have encountered.*

3. VERIFAS

In this section we describe the implementation of VERIFAS. We begin with a brief review of the theory developed in [17] that is relevant to the implementation.

3.1 Review of the Theory

The decidability and complexity results of [17] can be extended to HAS* by adapting the proofs and techniques developed there. We can show the following.

THEOREM 16. *Given a HAS* Γ and an LTL-FO formula φ for a task T in Γ, it is decidable in* EXPSPACE *whether Γ satisfies φ.*

We outline informally the roadmap to verification developed in [17], which is the starting point for the implementation. Let Γ be a HAS* and φ an LTL-FO formula for some task T of Γ. We would like to verify that every local run of T satisfies φ. Since there are generally infinitely many such local runs due so the unbounded data domain, and each run can be infinite, an exhaustive search is impossible. This problem is addressed in [17] by developing a symbolic representation of local runs. Intuitively, the symbolic representation has two main components:

(i) the *isomorphism type* of the artifact variables, describing symbolically the structure of the portion of the database reachable from the variables by navigating foreign keys

(ii) for each artifact relation and isomorphism type, the number of tuples in the relation that share that isomorphism type

Observe that because of (ii), the symbolic representation is not finite state. Indeed, (ii) requires maintaining a set of counters, which can grow unboundedly.

The heart of the proof in [17] is showing that it is sufficient to verify symbolic runs rather than actual runs. That is, for every LTL-FO formula φ, all local run of T satisfy φ iff all symbolic local runs of T satisfy φ. Then the verification algorithm checks that there is no symbolic local run of T violating φ (so satisfying $\neg\varphi$). The algorithm relies on a reduction to (repeated[4]) state reachability in Vector Addition Systems with States (VASS) [8]. Intuitively, VASS are finite-state automata augmented with non-negative counters that can be incremented and decremented (but not tested for zero). This turns out to be sufficient to capture the information described above. The states of the VASS correspond to the isomorphism types of the artifact variables, combined with states of the Büchi automaton needed to check satisfaction of $\neg\varphi$.

The above approach can be viewed as symbolically running the HAS* specification. Consider the example in Section 2. After the **TakeOrder** task is called and returned, one possible local run of **ProcessOrders** might impose a set of constraints $\{\texttt{item\_id} \neq \texttt{null}, \texttt{cust\_id} \neq \texttt{null}, \texttt{status} =$ "OrderPlaced", $\texttt{instock} =$ "Yes"$\}$ onto the artifact tuple of **ProcessOrders**. Now suppose the **CheckCredit** task is called. The local run can make the choice that the customer has good credit. Then when **CheckCredit** returns, the above set of constraints is updated with constraint $\{\texttt{cust\_id.record.status} =$ "Good"$\}$, which means that in the read-only database, the credit record referenced by `cust_id` via foreign key satisfies `status` = "Good". Next, suppose the *StoreOrder* service is applied in **ProcessOrders**. Then we symbolically store the current set of constraints by increasing its corresponding counter by 1. The set of constraints of the artifact tuple is reset to $\{\texttt{item\_id} = \texttt{null}, \texttt{cust\_id} = \texttt{null}, \texttt{status} =$ "Init"$\}$ as specified in the post-condition of *StoreOrder*.

Although decidability of verification can be shown as outlined above, implementation of an efficient verifier is challenging. The algorithm that directly translates the artifact specification and the LTL-FO property into VASS's and checks (repeated) reachability is impractical because the resulting VASS can have exponentially many states and counters in the input size, and state-of-the-art VASS tools can only handle a small number of counters ($<$100) [1]. To mitigate the inefficiency, VERIFAS never generates the whole VASS but instead lazily computes the symbolic representations on-the-fly. Thus, it only generates *reachable* symbolic states, whose number is usually much smaller. In addition, isomorphism types in the symbolic representation are replaced by *partial isomorphism types*, which store only the subset of constraints on the variables imposed by the current run, leaving the rest unspecified. This representation is not only more compact, but also results in a significantly smaller search space in practice.

In the rest of the section, we first introduce our revised symbolic representation based on partial isomorphism types. Next, we review the classic Karp-Miller algorithm adapted to the symbolic version of HAS* for solving state reachability problems. Three specialized optimizations are introduced to improve the performance. In addition, we show that our algorithm with the optimizations can be extended to solve the repeated state reachability problems so that full LTL-FO verification of infinite runs can be carried out. For clarity, the exposition in this section focuses on specifications with a single task. The actual implementation extends these techniques to the full model with arbitrary number of tasks.

3.2 Partial Isomorphism Types

We start with our symbolic representation of local runs with partial isomorphism types. Intuitively, a partial isomorphism type captures the necessary constraints imposed by the current run on the current artifact tuple and the read-only database. We start by defining *expressions*, which denote variables, constants and navigation via foreign keys from id variables or attributes. An expression is either:

[4]Repeated reachability is needed for infinite runs.

- a constant c occurring in Γ or φ, or
- a sequence $\xi_1.\xi_2.\ldots.\xi_m$, where ξ_1 is an id artifact variable x or an id attribute A of some artifact relation $\mathcal{S}$, ξ_2 is an attribute of $R \in \mathcal{DB}$ where $R.ID = \texttt{type}(\xi_1)$, and for each i, $2 \leq i < m$, ξ_i is a foreign key and ξ_{i+1} is an attribute in the relation referenced by ξ_i.

We denote by $\mathcal{E}$ the set of all expressions. Note that the length of expressions is bounded because of the acyclicity of the foreign keys, so $\mathcal{E}$ is finite.

We can now define partial isomorphism types.

DEFINITION 17. *A **partial isomorphism type** τ is an undirected graph over $\mathcal{E}$ with each edge labeled by $=$ or $\neq$, such that the equivalence relation $\sim$ over $\mathcal{E}$ induced by the edges labeled with $=$ satisfies:*

1. *for every $e, e' \in \mathcal{E}$ and every attribute A, if $e \sim e'$ and $\{e.A, e'.A\} \subseteq \mathcal{E}$ then $e.A \sim e'.A$, and*
2. *$(e_1, e_2, \neq) \in \tau$ implies that $e_1 \not\sim e_2$ and for every $e_1' \sim e_1$ and $e_2' \sim e_2$, $(e_1', e_2', \neq) \in \tau$.*

Intuitively, a partial isomorphism type keeps track of a set of "$=$" and "$\neq$" constraints and their implications among $\mathcal{E}$. Condition 1 guarantees satisfaction of the key and foreign key dependencies. Condition 2 guarantees that there is no contradiction among the $\neq$-edges and the $=$-edges. In addition, the connection between two expressions can also be "unknown" if they are not connected by an edge. The full isomorphism type can be viewed as a special case of partial isomorphism type where the undirected graph is complete. In the worst case, the total number of partial isomorphism types is no smaller than the number of full isomorphism types so using partial isomorphism types does not improve the complexity upper bound. In practice, however, since the number of constraints imposed by a run is likely to be small, using partial isomorphism types can greatly reduce the search space.

EXAMPLE 18. *Figure 5 shows two partial isomorphism types τ_1 (left) and τ_2 (right), where $R(\mathrm{ID}, A)$ is the only database relation and $\{x, y, z\}$ are 3 variables of type R.ID. Solid lines are $=$-edges and dashed lines are $\neq$-edges. In τ_1, (x, y) is connected with $=$ so the edge $(x.A, y.A, =)$ is enforced by the key dependency. Missing edges between $(x.A, z.A)$ and $(y.A, z.A)$ indicate these connections are "unknown". τ_2 is a full isomorphism type, which requires the graph to be complete so $(x.A, z.A)$, $(y.A, z.A)$ and all pairs between $\{x, y, z\}$ and $\{x.A, y.A, z.A\}$ must be connected by either $=$ or $\neq$. The $\neq$-edges between $\{x, y, z\}$ and $\{x.A, y.A, z.A\}$ are omitted in the figure for clarity.*

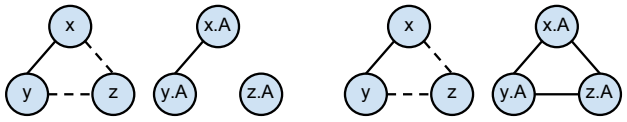

Figure 5: Partial and Full Isomorphism Types

We next define *partial symbolic instances*. Intuitively, a partial symbolic instance consists of a partial isomorphism type capturing the connections of the current tuple of $\bar{x}$, as well as, for the tuples present in the artifact relations, the represented isomorphism types t and the count of tuples sharing t.

DEFINITION 19. *A **partial symbolic instance** I is a tuple $(\tau, \bar{c})$ where τ is a partial isomorphism type and $\bar{c}$ is a vector of $\mathbb{N}$ where each dimension of $\bar{c}$ corresponds to a unique partial isomorphism type.*

It turns out that most of the dimensions of $\bar{c}$ equal 0 in practice, so in implementation we only materialize a list of those dimensions with positive counter values. We denote by $\texttt{pos}(\bar{c})$ the set $\{\tau_S | \bar{c}(\tau_S) > 0\}$.

Next, we define *symbolic transitions* among partial symbolic instances by applying internal services. First we need to define condition evaluation on partial isomorphism types. Given a partial isomorphism type τ, satisfaction of a condition ϕ in negation normal form[5] by τ, denoted $\tau \models \phi$, is defined as follows:

- $x \circ y$ holds in τ iff $(x, y, \circ) \in \tau$ for $\circ \in \{=, \neq\}$,
- for relation $R(ID, A_1, \ldots, A_m)$, $R(x, y_1, \ldots, y_m)$ holds in τ iff $(y_i, x.A_i, =) \in \tau$ for every $1 \leq i \leq m$,
- $\neg R(x, y_1, \ldots, y_m)$ holds in τ iff $(y_i, x.A_i, \neq) \in \tau$ for some $1 \leq i \leq m$, and
- Boolean combinations of conditions are standard.

Notice that $\tau \not\models \phi$ might be due to missing edges in τ but not because of inconsistent edges, so it is possible to satisfy ϕ by filling in the missing edges. This is captured by the notion of extension. We call τ' an *extension* of τ if $\tau \subseteq \tau'$ and τ' is consistent, meaning that the edges in τ' do not imply any contradiction of (in)equalities. We denote by $\texttt{eval}(\tau, \phi)$ the set of all *minimal extensions* τ' of τ such that $\tau' \models \phi$. Intuitively, $\texttt{eval}(\tau, \phi)$ contains partial isomorphism types obtained by augmenting τ with a minimal set of constraints to satisfy ϕ.

A symbolic transition is defined informally as follows (the full definition can be found in the extended version. To make a symbolic transition with a service $\sigma = (\pi, \psi, \bar{y}, \delta)$ from $I = (\tau, \bar{c})$ to $I' = (\tau', \bar{c}')$, we first extend the partial isomorphism type τ to a new partial isomorphism type τ_0 to satisfy the pre-condition π. Then the constraints on the propagated variables $\bar{y}$ are preserved by computing τ_1, the projection of τ_0 onto $\bar{y}$. Intuitively, the projection keeps only the expressions headed by variables in $\bar{y}$ and their connections. Finally, τ' is obtained by extending τ_1 to satisfy the post-condition ψ. If δ is an insertion, then the counter that corresponds to the partial isomorphism type of the inserted tuple is incremented. If δ is a retrieval, then a partial isomorphism type τ_S with positive count is chosen nondeterministically and its count is decremented. The new partial isomorphism type τ' is then extended with the constraints from τ_S. We denote by $\texttt{succ}(I)$ the set of possible successors of I by taking one symbolic transition with any service σ.

EXAMPLE 20. *Figure 6 shows an example of symbolic transition. The DB schema is that of Example 18. The variables are x, y of type R.ID and a non-ID variable z, with input variables $\{y, z\}$. The applied service is $\sigma = (\pi : R(x, z), \psi : x \neq y, \bar{y} : \{y, z\}, \delta : \{-\mathcal{S}(x, z)\})$. First, the pre-condition $R(x, z)$ is evaluated so edge $(x.A, z, =)$ is added (top-middle). Edge $(y.A, z, \neq)$ is also added so that the partial isomorphism type remains valid. Then variables $\{y, z\}$ are propagated, so the edges related to x or $x.A$ are removed (top-right). Next, we evaluate the post-condition $x \neq y$ so $(x, y, \neq)$ is added (bottom-right). Finally, a tuple from S is retrieved and overwrites $\{x, z\}$. Suppose the nondeterministically chosen τ_S contains a single edge $(x.A, z, =)$ (below the retrieve arrow). Then $\bar{c}(\tau_S)$ is decremented and τ_S is merged into the final partial isomorphism type (bottom left). Note that if σ contains an insertion of $+S(x, z)$ in δ instead of a retrieval, then the subgraph of τ_0 (top-middle) projected to $\{x, z\}$ is inserted to S. The corresponding counter in $\bar{c}$ will be incremented by 1.*

With symbolic transitions in place, verification works as follows. Informally, given a single-task HAS* Γ and a LTL-FO property φ, one can check whether $\Gamma \models \varphi$ by constructing a new HAS* Γ' obtained by combining Γ with conditions in φ and the Büchi automaton $B_{\neg\varphi}$ built from $\neg\varphi$. We can show that deciding whether

[5]Negations are pushed down to leaf atoms.

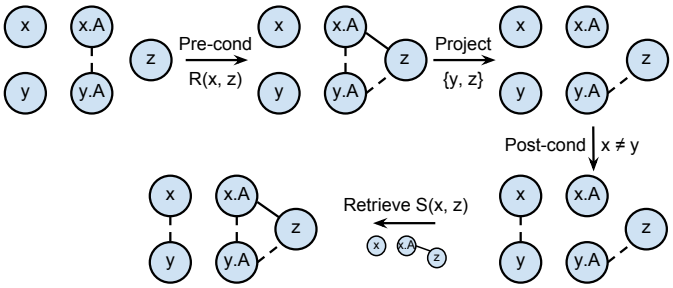

Figure 6: Symbolic Transition

$\Gamma \not\models \varphi$ reduces to checking whether an accepting state of $B_{\neg\varphi}$ is repeatedly reachable in Γ'. Verification therefore amounts to solving the following problem:

PROBLEM 21. *(Symbolic Repeated Reachability, or SRR) Given a HAS* Γ, an initial partial symbolic state I_0, and a condition ϕ, is there a partial symbolic run $\{I_i\}_{0\leq i\leq m<n}$ of Γ such that $I_{i+1} \in \texttt{succ}(I_i)$ for every $i \geq 0$, $\tau_m = \tau_n$, $\bar{c}_m \leq \bar{c}_n$ and $\tau_n \models \phi$?*

The condition ϕ above simply states that $B_{\neg\varphi}$ is in one of its accepting states.

3.3 The Classic Karp-Miller Algorithm

The SRR Problem defines an infinite search space due to the unbounded counter values, so reduction to finite-state model checking is not possible. Adapting the theory developed in [17] from symbolic representation based on isomorphism types to symbolic representation based on partial isomorphism types, we can show that the symbolic transitions defined in Section 3.2 can be modeled as a VASS whose states are the partial symbolic instances of Definition 19. Consequently, The SRR problem reduces to testing (repeated) state reachability in this VASS. The benefit of the new approach is that this VASS is likely to have much fewer states and counters than the one defined in [17], because our search materializes partial isomorphism types parsimoniously, by lazily expanding the current partial type c using the (typically few) constraints of the symbolic transition to obtain a successor s. The transition from c to s concisely represents all the transitions from full-type expansions of c to full-type expansions of s (exponentially many in the number of "unknown" connections in c and s), which in the worst case would be individually explored by the algorithm of [17].

The VERIFAS implementation of the (repeated) state reachability is based on a series of optimizations to the classic Karp-Miller algorithm [29]. We describe the original Karp-Miller algorithm first, addressing the optimizations subsequently.

The Karp-Miller algorithm constructs a finite representation which over-approximates the entire (potentially infinite) reachable VASS state space, called a "coverability set" of states [21]. Any coverability set captures sufficient information about the global state space to support global reasoning tasks, including repeated reachability. In our context, the VASS states are the partial symbolic instances (PSIs) and a *coverability* set is a finite set $\mathcal{I}$ of PSIs, each reachable from the initial PSI I_0, such that for every reachable PSI $I = (\tau, \bar{c})$, there exists $I' = (\tau', \bar{c}') \in \mathcal{I}$ with $\tau = \tau'$ and $\bar{c} \leq \bar{c}'$. We say that I' *covers* I, denoted $I \leq I'$. To represent counters that can increase forever, the coverability set also allows an extension of PSIs in which some of the counters can equal ω. Recall that the ordinal ω is a special constant where $n < \omega$ for all $n \in \mathbb{N}$, $\omega \leq \omega$ and $\omega \pm 1 = \omega$.

Since the coverability set $\mathcal{I}$ is finite, we can effectively extract from it the reachable τ_n's that satisfy the condition ϕ (referring to the notation of the SRR problem). To test whether τ_n is repeatedly reachable, we can show that I_n is repeatedly reachable iff I_n is contained in a cycle consisting of only states in $\mathcal{I}$ (proved in [18]). As a result, the repeatedly reachable τ_n's can be found by constructing the transition graph among $\mathcal{I}$ and computing its *strongly connected components*. A partial isomorphism type τ is repeatedly reachable if its corresponding PSI I is included in a component containing a non-trivial cycle.

The Karp-Miller algorithm searches for a coverability set by materializing a finite part of the (potentially infinite) VASS transition graph starting from the initial state and executing transitions, pruning transitions to states that are covered by already materialized states. The resulting transition subgraph is traditionally called the *Karp-Miller tree* (despite the fact that it is actually a DAG).

In the notation of the SRR problem, note that if at least one counter value strictly increases from I_m to I_n ($\bar{c}^i_m < \bar{c}^i_n$ for some dimension i), then the sequence of transitions from m to n can repeat indefinitely, periodically reaching states with the same partial isomorphism type τ_n, but with ever-increasing associated counter values in dimension i (there are infinitely many such states). In the limit, the counter value becomes ω so a coverability set must include a state $(\tau_n, \bar{c})$ with $\bar{c}^i = \omega$, covering these infinitely many states.

Finite construction of the tree is possible due to a special *accelerate* operation that skips directly to a state with ω-valued counters, avoiding the construction of the infinitely many states it covers. Adapted to our context, when the algorithm detects a path $\{I_i\}_{0\leq i\leq m<n}$ in the tree where $I_m \leq I_n$, the accelerate operation replaces in I_n the values of $\bar{c}_n(\tau_S)$ with ω for every τ_S where $\bar{c}_m(\tau_S) < \bar{c}_n(\tau_S)$.

We outline the details in Algorithm 1, which outputs the Karp-Miller tree $\mathcal{T}$. We denote by $\texttt{ancestors}(I)$ the set of ancestors of I in $\mathcal{T}$. Given a set $\mathcal{I}$ of states and a state $I' = (\tau', \bar{c}')$, the accelerate function is defined as $\texttt{accel}(\mathcal{I}, I') = (\tau', \bar{c}'')$ where for every τ_S, $\bar{c}''(\tau_S) = \omega$ if there exists $(\tau, \bar{c}) \in \mathcal{I}$ such that $\tau = \tau'$, $\bar{c} \leq \bar{c}'$ and $\bar{c}(\tau_S) < \bar{c}'(\tau_S)$. Otherwise, $\bar{c}''(\tau_S) = \bar{c}'(\tau_S)$.

Algorithm 1: Karp-Miller Tree Search Algorithm

input : Initial instance I_0
output : $\mathcal{T}$, the Karp-Miller tree
variables: W, set of states waiting to be explored

```
W ← {I_0}, T ← ({I_0}, ∅);
while W ≠ ∅ do
    Remove a state I from W;
    for I' ∈ succ(I) do
        I'' ← accel(ancestors(I), I');
        if I'' ∉ T ∨ I'' ∈ W then
            Add edge (I, I'') to T;
            W ← W ∪ {I''};
Return T;
```

3.4 Optimization with Monotone Pruning

The original Karp-Miller algorithm is well-known to be inefficient in practice due to state explosion. To improve performance, various techniques have been proposed. The main technique we adopted in VERIFAS is based on pruning the Karp-Miller tree by monotonicity. Intuitively, when a new state $I = (\tau, \bar{c})$ is generated during the search, if there exists a visited state I' where $I \leq I'$, then I can be discarded because for every state $\tilde{I}$ reachable from I, there exists a reachable state $\tilde{I}'$ starting from I' such that $\tilde{I} \leq \tilde{I}'$ by applying the same sequence of transitions that leads I to $\tilde{I}$. For the same reason, if $I \geq I'$, then I' and its descendants can be pruned

from the tree. However, correctness of pruning is sensitive to the order of application of these rules (for example, as illustrated in [22], application of the rules in a breadth-first order may lead to incompleteness). The problem of how to apply the rules without losing completeness was studied in [22, 38] and we adopt the approach in [38]. More specifically, Algorithm 1 is extended by keeping track of a set `act` of "active" states and adding the following changes:

- Initialize `act` with $\{I_0\}$;
- In line 3, choose the state from $W \cap$ `act`;
- In line 5, `accel` is applied on `ancestors`$(I) \cap$ `act`;
- In line 8, I'' is not added to W if there exists $\hat{I} \in$ `act` such that $I'' \leq \hat{I}$;
- When I'' is added to W, remove from `act` every state $\hat{I}$ and its descendants for $\hat{I} \leq I''$ and $\hat{I}$ is either active or not an ancestor of I''. Add I'' to `act`.

3.5 A Novel, More Aggressive Pruning

We generalize the comparison relation $\leq$ of partial symbolic instances to achieve more aggressive pruning of the explored transitions. The novel comparison is based on the insight that a state I can be pruned in favor of I' as long as every partial isomorphism type reachable from I is also reachable from I'. So $I = (\tau, \bar{c})$ can be pruned by $I' = (\tau', \bar{c}')$ if τ' is "less restrictive" than τ (or τ implies τ'), and for every occurrence of τ_S in $\bar{c}$, there exists a corresponding occurrence of τ'_S in $\bar{c}'$ such that τ'_S is "less restrictive" than τ_S. Formally, given partial isomorphism types τ and τ', τ implies τ', denoted as $\tau \models \tau'$, iff $\tau' \subseteq \tau$. We replace the coverage relation $\leq$ on partial symbolic instances with a new binary relation $\preceq$ as follows.

Definition 22. *Given two partial symbolic states $I = (\tau, \bar{c})$ and $I' = (\tau', \bar{c}')$, $I \preceq I'$ iff $\tau \models \tau'$ and there exists f : `pos`$(\bar{c}) \times$ `pos`$(\bar{c}') \mapsto \mathbb{N} \cup \{\omega\}$ such that*

- *$f(\tau_S, \tau'_S) > 0$ only when $\tau_S \models \tau'_S$,*
- *for every τ_S, $\sum_{\tau'_S} f(\tau_S, \tau'_S) = \bar{c}(\tau_S)$ and*
- *for every τ'_S, $\sum_{\tau_S} f(\tau_S, \tau'_S) \leq \bar{c}'(\tau'_S)$.*

Intuitively, f describes a one-to-one mapping from tuples stored in the artifact relations in I to tuples in I'. $f(\tau_S, \tau'_S) = k$ means that there are k tuples in I of partial isomorphism type τ_S that are mapped to k tuples in I' of type τ'_S. The condition $\tau_S \models \tau'_S$ guarantees that each tuple in I is mapped to one in I' of a less restrictive type.

Example 23. *Consider the two PSIs $I = (\tau, \bar{c} = \{\tau_a : 2, \tau_b : 2\})$ (left) and $I' = (\tau', \bar{c}' = \{\tau_a : 3, \tau_b : 1\})$ (right) shown in Figure 7. Since $\tau \neq \tau'$ and $\bar{c}(\tau_b) > \bar{c}'(\tau_b)$, $I \leq I'$ does not hold. However, any sequence of symbolic transitions applicable starting from I can also be applied starting from I', because if the conditions imposed by these transitions do not conflict with those in I, then they won't conflict with the subset thereof in I'. Consequently, I can be pruned if I' is found during the search. This fact is detected by $\preceq$: $I \preceq I'$ holds since $\tau \models \tau'$ and we can construct f as $f(\tau_a, \tau_a) = 2$ and $f(\tau_b, \tau_b) = f(\tau_b, \tau_a) = 1$ since $\tau_b \models \tau_a$.*

Note that one can efficiently test whether $I \preceq I'$ by reduction to the Max-Flow problem over a flow graph F with node set `pos`$(\bar{c}) \cup$ `pos`$(\bar{c}') \cup \{s, t\}$, with s a source node and t a sink node. For every $\tau_S \in$ `pos`$(\bar{c})$, there is an edge from s to τ_S with capacity $\bar{c}(\tau_S)$. For every $\tau'_S \in$ `pos`$(\bar{c}')$, there is an edge from τ'_S to t with capacity $\bar{c}'(\tau_S)$. For every pair of τ_S, τ'_S, there is an edge from τ_S to τ'_S with capacity ∞ if $\tau'_S \models \tau_S$. We can show that F has a max-flow equal to $\sum_{\tau_S} \bar{c}(\tau_S)$ if and only if $I \preceq I'$.

The same idea can also be applied to the `accel` function. Formally, given $\mathcal{I}$ and $I' = (\tau', \bar{c}')$, the new accelerate function

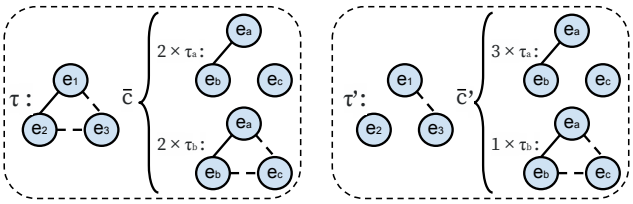

Figure 7: Illustration of $\preceq$

`accel`$(\mathcal{I}, I') = (\tau', \bar{c}'')$ where $\bar{c}''(\tau'_S) = \omega$ if there exists $I \in \mathcal{I}$ such that $I \preceq I'$ and there exists mapping f satisfying the conditions in Definition 22 and $\sum_{\tau_S} f(\tau_S, \tau'_S) < \bar{c}'(\tau'_S)$. Otherwise $\bar{c}''(\tau'_S) = \bar{c}'(\tau'_S)$.

3.6 Data Structure Support

The above optimization relies on two important operations applied every time a new state is explored: given the set of active states `act` and a partial symbolic state I, (1) compute the set $\{I' | I' \preceq I \wedge I' \in$ `act`$\}$ and (2) check whether there exists $I' \in$ `act` such that $I \preceq I'$. As each test of $\preceq$ might require an expensive operation of computing the max-flow, when |`act`| is large, checking whether $I' \preceq I$ (or $I \preceq I'$) for every $I' \in$ `act` would be too time-consuming.

We start with the simple case where $\bar{c} = \bar{0}$ in all I's. Then to test whether $I \preceq I'$ for $I = (\tau, \bar{c})$ and $I' = (\tau', \bar{c}')$ is to test whether $\tau' \subseteq \tau$. When the partial isomorphism types are stored as sets of edges, we can accelerate the two operations with data structures that support fast subset (superset) queries: given a collection $\mathcal{C}$ of sets and a query set q, find all sets in $\mathcal{C}$ that are subsets (supersets) of q. The standard solutions are to use Tries for superset queries [39] and Inverted Lists for subset queries [33].

The same idea can be applied to the general case where $\bar{c} \geq 0$ to obtain over-approximations of the precise results. Given $I = (\tau, \bar{c})$, we let $E(I)$ be the set of edges in τ or any τ_S where $\bar{c}(\tau_S) > 0$. Then given I and I', $I \preceq I'$ implies $E(I') \subseteq E(I)$. We build the Trie and Inverted Lists indices such that for a given query I, they return a candidate set $\mathcal{I}^{\preceq}$ and a candidate set $\mathcal{I}^{\succeq}$ where $\mathcal{I}^{\preceq}$ contains all I' from `act` such that $E(I') \subseteq E(I)$ and $\mathcal{I}^{\succeq}$ contains all I' such that $E(I) \subseteq E(I')$. Then it suffices to test each member in the candidate sets for $I \preceq I'$ and $I \succeq I'$ to obtain the precise results of operations (1) and (2).

3.7 Optimization with Static Analysis

Next, we introduce our optimization based on static analysis. At a high level, we notice that in real workflow examples, some constraints in conditions of the specification and the property are irrelevant to the result of verification because they can never cause violations when conditions are evaluated in a symbolic run. Such conditions can be ignored to reduce the number of symbolic states. For example, for a constraint $x = y$ in the specification, if $x \neq y$ does not appear anywhere else and cannot be implied by other constraints, then $x = y$ can be safely removed from any partial isomorphism types without affecting the result of the verification algorithm. Our goal is to detect all such constraints by statically analyzing the HAS* and the LTL-FO property. Specifically, we analyze the constraint graph consisting of all possible "=" and "≠" constraints that can potentially be added to any partial isomorphism types in symbolic transitions of the HAS* Γ or when checking condition ϕ (refer to the notation in the SRR problem).

Definition 24. *The constraint graph G of (Γ, ϕ) is a labeled undirected graph over the set of all expressions $\mathcal{E}$ with the following*

edges. For every atom a that appears in a condition of Γ or ϕ in negation normal form, if a is

- $(x = y)$*, then G contains $(x.w, y.w, =)$ for all sequences w where $\{x.w, y.w\} \subseteq \mathcal{E}$,*
- $(x \neq y)$*, then G contains $(x, y, \neq)$,*
- $R(x, y_1, \ldots, y_m)$*, then G contains $(x.A_i.w, y_i.w, =)$ for all i and sequences w where $\{x.A_i.w, y_i.w\} \subseteq \mathcal{E}$, and*
- $\neg R(x, y_1, \ldots, y_m)$*, then G contains $(x.A_i, y_i, \neq)$ for all i.*

For any subgraph G' of G, G' is consistent *if the edges in G' do not imply any contradiction, meaning that there is no path of $=$-edges connecting two distinct constants or two expressions connected by an $\neq$-edge.*

An edge e of G is non-violating *if for every consistent subgraph G', $G' \cup \{e\}$ is also consistent.*

Intuitively, by collecting the edges described above, the constraint graph G becomes an over-approximation of the reachable partial isomorphism types. Thus any edge in G that is non-violating is also non-violating in any reachable partial isomorphism type. So our goal is to find all the non-violating edges in G, since they can be ignored in partial isomorphism types to reduce the size of the search space.

Non-violating edges can be identified efficiently in polynomial time. Specifically, an edge $(u, v, \neq)$ is non-violating if u and v belong to different connected components of $=$-edges of G. An $=$-edge e is non-violating if there is no path $u \longrightarrow v$ of $=$-edges containing e for any $(u, v, \neq) \in G$ or (u, v) being two distinct constants. This can be checked efficiently by computing the biconnected components of the $=$-edges [44]. We omit the details here.

Example 25. *Consider the two constraint graphs G_1 and G_2 in Figure 8. In G_1 (left), (e_3, e_5) is a non-violating $\neq$-edge because e_3 and e_5 belong to two different connected components of $=$-edges ($\{e_1, e_2, e_3, e_4\}$ and $\{e_5, e_6, e_7\}$ respectively). In G_2 (right), (e_3, e_5) is a non-violating $=$-edge because (e_3, e_5) is not on any simple path of $=$-edges connecting the two ends of any $\neq$-edges (i.e. (e_2, e_3) and (e_5, e_6)).*

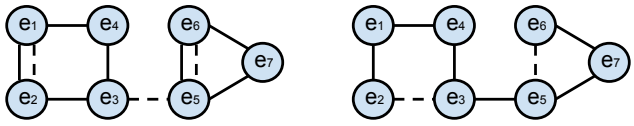

Figure 8: Non-violating Edges

3.8 Extension to Repeated-Reachability

Recall from Section 3.2 that providing full support for verifying LTL-FO properties requires solving the repeated state reachability problem. It is well-known that for VASS, the coverability set $\mathcal{I}$ extracted from the tree $\mathcal{T}$ constructed by the classic Karp-Miller algorithm can be used to identify the repeatedly reachable partial isomorphism types (see Section 3.3). The same idea can be extended to the Karp-Miller algorithm with monotone pruning (Section 3.4), since the algorithm is guaranteed to construct a coverability set.

However, the Karp-Miller algorithm equipped with our $\preceq$-based pruning (Section 3.5) explores fewer states due to the more aggressive pruning, and it turns out that the resulting coverability set $\mathcal{I}^{\preceq}$ is incomplete to determine whether a state is repeatedly reachable. We can no longer guarantee that a repeatedly reachable state is only contained in a cycle of states in $\mathcal{I}^{\preceq}$. We can nevertheless show that the completeness of the search for repeatedly reachable states can be restored by developing our own extraction technique which compensates for the overly aggressive $\preceq$-based pruning. The technical development is subtle and relegated to the extended version [18]. As confirmed by our experimental results, the additional overhead is acceptable.

4. EXPERIMENTAL EVALUATION

We evaluated the performance of VERIFAS using both real-world and synthetic artifact specifications.

4.1 Setup and Benchmark

The Real Set We built an artifact system benchmark by rewriting in HAS* a sample of real-world BPMN workflows published at the official BPMN website [2], which provides 36 workflows of non-trivial size. To rewrite these workflows in HAS*, we manually added the database schema, artifact variables/relations, and services for updating the data. HAS* is sufficiently expressive to specify 32 of the 36 BPMN workflows. The remaining 4 cannot be expressed in HAS* because they involve computing aggregate functions or updating unboundedly many tuples of the artifact relations, which is not supported in the current model. We will consider such features in our future work.

The Synthetic Set Since we wished to stress-test VERIFAS, we also randomly generated a set of HAS* specifications of increasing complexity. All components of each specification, including DB schema, task hierarchy, variables, services and conditions, were generated fully at random for a certain size. We provide in the full paper [18] more details on how each specification is generated. Those with empty state space due to unsatisfiable conditions were removed from the benchmark. Table 1 shows some statistics of the two sets of specifications. (#Relations, #Tasks, etc. are averages over the real / synthetic sets of workflows.)

Table 1: Statistics of the Two Sets of Workflows

Dataset	Size	#Relations	#Tasks	#Variables	#Services
Real	32	3.563	3.219	20.63	11.59
Synthetic	120	5	5	75	75

LTL-FO Properties On each workflow of both sets, we run our verifier on a collection of 12 LTL-FO properties of the root task constructed using templates of real propositional LTL properties. The LTL properties are all the 11 examples of safety, liveness and fairness properties collected from a standard reference paper [42] and an additional property `False` used as a baseline when comparing the performance of VERIFAS on different classes of LTL-FO properties. We list all the templates of LTL properties in Table 4. To see why we choose `False` as the baseline property, recall from Section 3 that the verifier's running time is mainly determined by the size of the reachable symbolic state space (VERIFAS first computes all reachable symbolic states –represented by the coverability set– then identifies the repeatedly-reachable ones). The reachable symbolic state space can be conceptualized as the cross-product between the reachable symbolic state space of the HAS* specification (absent any property) and the Büchi automaton of the property. When the LTL-FO property is `False`, the generated Büchi automaton is of the simplest form (a single accepting state within a loop), so it has no impact on the cross-product size, unlike more complex properties.

In each workflow, we generate an LTL-FO property for each template by replacing the propositions with FO conditions chosen from the pre-and-post conditions and their sub-formulas. Note that by doing so, the generated LTL-FO properties on the real workflows are combinations of real propositional LTL properties and real FO conditions, and so are close to real-world LTL-FO properties.

Baseline We compare VERIFAS with a simpler implementation built on top of Spin, a widely used software verification tool [27]. Building such a verifier is a challenging task in its own right since Spin is essentially a finite-state model checking tool and hence is incapable of handling data of unbounded size, present in the HAS*

Table 2: Average Elapsed Time and Number of Failed Runs (#Fail) due to Timeout or Memory Overflow

Verifier	Real		Synthetic	
	Avg(Time)	#Fail	Avg(Time)	#Fail
Spin-Opt	2.97s	3	83.983s	440
VERIFAS-NoSet	**.229s**	0	**6.983s**	19
VERIFAS	**.245s**	0	**11.01s**	16

model. We managed to build a Spin-based verifier supporting a restricted version of our model that does not handle updatable artifact relations. As the read-only database can still have unbounded size and domain, the Spin-based implementation requires nontrivial translations and optimizations, which are presented in detail in [32].

Platform We implemented both verifiers in C++ with Spin version 6.4.6 for the Spin-based verifier. All experiments were performed on a Linux server with a quad-core Intel i7-2600 CPU and 16G memory. For each specification, we ran our verifiers to test each of the 12 generated LTL-FO properties, resulting in 384 runs for the real set and 1440 runs for the synthetic set. Towards fair comparison, since the Spin-based verifier (Spin-Opt) cannot handle artifact relations, in addition to running our full verifier (VERIFAS), we also ran it with artifact relations ignored (VERIFAS-NoSet). The timeout limit of each run was set to 10 minutes and the memory limit was set to 8G.

4.2 Experimental Results

Performance Table 2 shows the results on both sets of workflows. The Spin-based verifier achieves acceptable performance in the real set, with an average elapsed time of a few seconds and only 3 timeouts. However, it failed in a large number of runs (440/1440) in the stress-test using synthetic specifications. On the other hand, both VERIFAS and VERIFAS-NoSet achieve average running times within 0.3 second and no timeout on the real set, and the average running time is within seconds on the synthetic set, with only 19 timeouts over 1440 runs. The presence of artifact relations introduced an acceptable amount of performance overhead, which was negligible in the real set and less than 60% in the synthetic set. Compared with the Spin-based verifier, VERIFAS is >10x faster in average running time and scales significantly better with workflow complexity.

The timeout runs on the synthetic workflows are all due to state explosion with a state space of size $\sim 3 \times 10^4$. The reason is that though unlikely in practice, it is still possible that the reached partial isomorphism types can degenerate to full isomorphism types, and in this case our state-pruning optimization does not reduce the number of reached states.

Cyclomatic Complexity To better understand the scalability of VERIFAS, we also measured verification time as a function of workflow complexity, adopting a metric called *cyclomatic complexity*, which is widely used in measuring complexity of program modules [48]. For a program P with control-flow graph $G(V, E)$, the cyclomatic complexity of P equals $|E| - |V| + 2$. We adapt this measure to HAS* specifications as follows. Given a HAS* specification $\mathcal{A}$, a control flow graph of $\mathcal{A}$ can be obtained by selecting a task T of $\mathcal{A}$ and a non-id variable $x \in \bar{x}^T$ and projecting all services of T onto $\{x\}$. The resulting services contain only x and constants and thus can be viewed as a transition graph with x as the state variable. The cyclomatic complexity of $\mathcal{A}$, denoted as $M(\mathcal{A})$, is defined as the maximum cyclomatic complexity over all the possible control-flow graphs of $\mathcal{A}$ (corresponding to all possible projections).

Figure 9 shows that the verification time increases exponentially with the cyclomatic complexity, thus confirming the pertinence of the measure to predicting verification complexity, where the verification time of a workflow is measured by the average running time over all the runs of its LTL-FO properties. According to [48]'s recommendation, for a program to remain readable and testable, its cyclomatic complexity should not exceed 15. Among all the 138 workflows with cyclomatic complexity at most 15, VERIFAS successfully verified 130/138 (∼94%) of them within 10s and only 4 instances have timeout runs (marked as hollow triangles in Figure 9). For specifications with complexity above 15, only 2/14 instances have timeout runs.

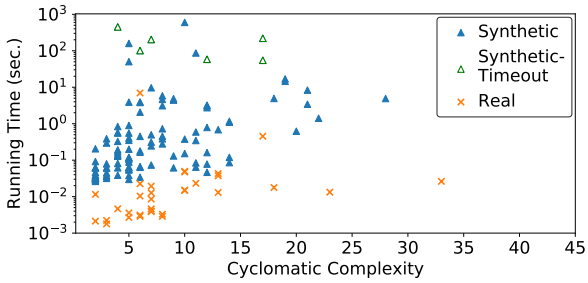

Figure 9: Average Running Time vs. Cyclomatic Complexity

Typically, for the same cyclomatic complexity, the real workflows can be verified faster compared to the synthetic workflows. This is because the search space of the synthetic workflows is likely to be larger because there are more variables and transitions.

Impact of Optimizations We studied the effect of our optimization techniques: state pruning (SP, Section 3.5), data structure support (DSS, Section 3.6), and static analysis (SA, Section 3.7). For each technique, we reran the experiment with the optimization turned off and measured the speedup by comparing the elapsed verification time with the original elapsed time. Table 3 shows the average speedups of each optimization on both datasets. Since some instances have extreme speedups (over 10,000x), simply averaging could be misleading, so we also present the trimmed averages of the speedups (i.e. removing the top/bottom 5% speedups before averaging) to exclude the extreme values.

Table 3 shows that the effect of state pruning is the most significant in both sets of workflows, with an average (trimmed) speedup of ∼25x and ∼127x in the real and synthetic set respectively. The static analysis optimization is more effective in the real set (1.4x improvement) but its effect in the synthetic set is less pronounced. It creates a small amount (7%) of overhead in most cases, but significantly improves the running time of a single instance, resulting in the huge gap between the normal average speedup and the trimmed average speedup. The explanation to this phenomenon is that the workflows in the real set are more "sparse" in general, which means there are fewer comparisons within a subset of variables so a larger number of comparisons can be pruned by static analysis. Finally, the data-structure support provides ∼1.2x and ∼1.6x average speedup in each set respectively. Not surprisingly, the optimization becomes more effective as the size of the state space increases.

Table 3: Mean and Trimmed Mean (5%) of Speedups

Dataset	SP		SA		DSS	
	Mean	Trim.	Mean	Trim.	Mean	Trim.
Real	1586.54x	24.69x	1.80x	1.41x	1.87x	1.24x
Synthetic	322.03x	127.35x	28.78x	0.93x	2.72x	1.58x

Overhead of Repeated-Reachability We evaluated the overhead of computing the set of repeatedly-reachable states from the coverability set (Section 3.8) by repeating the experiment with the re-

peated reachability module turned off. Compared with the turned off version, the full verifier has an average overhead of **19.03%** on the real set and **13.55%** overhead on the synthetic set (overheads are computed over the non-timed-out runs).

Effect of Different Classes of LTL-FO Properties Finally, we evaluate how the structure of LTL-FO properties affects the performance of VERIFAS. Table 4 lists all the LTL templates used in generating the LTL-FO properties and their intuitive meaning stated in [42]. For each template and for each set of workflows, we measure the average running time over all the runs with LTL-FO properties generated using the template. Table 4 shows that for each class of properties, the average running time is within 2x of the average running time for the simplest non-trivial property `False`. This is much better than the theoretical upper bound, which is linear in size of the Büchi automaton of the LTL formula. Some properties even have a shorter running time because, although the space of partial symbolic instances is enlarged by the Büchi automaton, more of the states may become unreachable due to the additional constrains imposed by the LTL-FO property.

Table 4: Average Running Time of Verifying Different Classes of LTL-FO Properties

Templates for LTL-FO	Meaning	Real	Synthetic
`False`	Baseline	0.26s	10.13s
$\mathbf{G}\varphi$	Safety	0.28s	10.26s
$(\neg\varphi\ \mathbf{U}\ \psi)$	Safety	0.28s	16.13s
$(\neg\varphi\mathbf{U}\psi) \wedge \mathbf{G}(\varphi \rightarrow \mathbf{X}(\neg\varphi\mathbf{U}\psi))$	Safety	0.30s	10.79s
$\mathbf{G}(\varphi \rightarrow (\psi \vee \mathbf{X}\psi \vee \mathbf{XX}\psi))$	Safety	0.29s	12.07s
$\mathbf{G}(\varphi \vee \mathbf{G}(\neg\varphi))$	Safety	0.30s	12.17s
$\mathbf{G}(\varphi \rightarrow \mathbf{F}\psi)$	Liveness	0.29s	16.81s
$\mathbf{F}\varphi$	Liveness	0.02s	6.44s
$\mathbf{GF}\varphi \rightarrow \mathbf{GF}\psi$	Fairness	0.30s	14.09s
$\mathbf{GF}\varphi$	Fairness	0.28s	6.91s
$\mathbf{G}(\varphi \vee \mathbf{G}\psi)$	Fairness	0.05s	9.64s
$\mathbf{FG}\varphi \rightarrow \mathbf{GF}\psi$	Fairness	0.28s	6.75s

5. RELATED WORK

The artifact verification problem has previously been studied mainly from a theoretical perspective. As mentioned in Section 1, fully automatic artifact verification is a challenging problem due to the presence of unbounded data. To deal with the resulting infinite-state system, we developed in [16] 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 [11] we extended our approach to allow for database dependencies and numeric data testable by arithmetic constraints. The symbolic approach developed in [16] and its extension to HAS [17] provides the theoretical foundation for VERIFAS.

Another theoretical line of work considers the verification problem for runs starting from a *fixed* initial database. During the run, the database may evolve via updates, insertions and deletions. Since inputs may contain fresh values from an infinite domain, this verification variant remains infinite-state. The property languages are fragments of first-order-extended μ-calculus [13]. Decidability results are based on sufficient syntactic restrictions [13, 26, 9]. [5] derives decidability of the verification variant by also disallowing unbounded accumulation of input values, but this condition is postulated as a semantic property (shown undecidable in [26]). [3] takes a different approach, in which decidability is obtained for *recency-bounded* artifacts, in which only recently introduced values are retained in the current data.

On the practical side of artifact verification, [15] considers the verification of business processes specified in a Petri-net-based model extended with data and process components, in the spirit of the theoretical work of [40, 4, 31, 41], which considers extending Petri nets with data-carrying tokens. The verifier of [15] differs fundamentally from ours in that properties are checked only for a given initial database. In contrast, our verifier checks that *all* runs satisfy given properties *regardless* of the initial underlying database. [25] and its prior work [23, 24] implemented a verifier for artifact systems specified directly in the GSM model. While the above models are expressive, the verifiers require restrictions of the models strongly limiting modeling power [23], or predicate abstraction resulting in loss of soundness and/or completeness [24, 25]. Lastly, the properties verified in [24, 25] focus on temporal-epistemic properties in a multi-agent finite-state system. Thus, the verifiers in these works have a different focus and are incomparable to ours.

Practical verification has also been studied in business process management (see [45] for a survey). The considered models are mostly process-driven (BPMN, Workflow-Net, UML etc.), with the business-relevant data abstracted away. The implementation of a verifier for data-driven web applications was studied in [19, 20]. The model is similar in flavor to the artifact model, but much less expressive. The verification approach developed there is not applicable to HAS*, which requires substantially new tools and techniques. Finally, our own work on building a verifier based on Spin was discussed in Section 1, and compared to VERIFAS in Section 4.

6. CONCLUSION

We presented the implementation of VERIFAS, an efficient verifier of temporal properties for data-driven workflows specified in HAS*, a variant of the Hierarchical Artifact System model studied theoretically in [17]. HAS* is inspired by the Business Artifacts framework introduced by IBM [28] and incorporated in OMG's CMMN standard [34, 36].

While the verification problem is EXPSPACE-complete (see extended version [18]) our experiments show that the theoretical worst case is unlikely in practice and that verification is eminently feasible. Indeed, VERIFAS achieves excellent performance (verification within seconds) on a practically relevant class of real-world and synthetic workflows (those with cyclomatic complexity in the range recommended by good software engineering practice), and a set of representative properties. The good performance of VERIFAS is due to an adaptation of our symbolic verification techniques developed in [17], coupled with the classic Karp-Miller algorithm accelerated with an array of nontrivial novel optimizations.

We also compared VERIFAS to a verifier we built on top of the widely used model checking tool Spin. VERIFAS not only applies to a much broader class of artifacts but also outperforms the Spin-based verifier by over one order of magnitude even on the simple artifacts the Spin-based verifier is able to handle. To the best of our knowledge, VERIFAS is the first implementation of practical significance of an artifact verifier with full support for unbounded data. In future work, we plan to extend VERIFAS to support a more expressive model that captures true parallelism, aggregate functions and arithmetic.

Acknowledgement We are grateful to the anonymous reviewers for their thorough reports and many suggestions that have greatly improved the paper. This work was supported in part by the National Science Foundation under award IIS-1422375.

7. REFERENCES

[1] MIST - a safety checker for Petri nets and extensions. `https://github.com/pierreganty/mist/wiki`. Accessed: 2017-04-13.

[2] Object management group business process model and notation. `http://www.bpmn.org/`. Accessed: 2017-03-01.

[3] P. A. Abdulla, C. Aiswarya, M. F. Atig, M. Montali, and O. Rezine. Recency-bounded verification of dynamic database-driven systems. In *PODS*, pages 195–210, 2016.

[4] E. Badouel, L. Hélouët, and C. Morvan. Petri nets with semi-structured data. In *Petri Nets*, 2015.

[5] F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of GSM-based artifact-centric systems through finite abstraction. In *ICSOC*, pages 17–31, 2012.

[6] 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.

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

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

[9] D. Calvanese, G. Delzanno, and M. Montali. Verification of relational multiagent systems with data types. In *AAAI*, pages 2031–2037, 2015.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] R. De Masellis, C. Di Francescomarino, C. Ghidini, M. Montali, and S. Tessaris. Add data into business process verification: Bridging the gap between theory and practice. In *AAAI*, 2017.

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

[17] A. Deutsch, Y. Li, and V. Vianu. Verification of hierarchical artifact systems. In *PODS*, pages 179–194, 2016.

[18] A. Deutsch, Y. Li, and V. Vianu. VERIFAS: A practical verifier for artifact systems (extended version). *arXiv preprint*, arXiv:1705.10007, 2017.

[19] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD*, pages 539–550, 2005.

[20] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. A system for specification and verification of interactive, data-driven web applications. In *SIGMOD*, pages 772–774, 2006.

[21] A. Finkel. The minimal coverability graph for Petri nets. *Advances in Petri Nets 1993*, pages 210–243, 1993.

[22] G. Geeraerts, J.-F. Raskin, and L. Van Begin. On the efficient computation of the minimal coverability set of petri nets. *International Journal of Foundations of Computer Science*, 21(02):135–165, 2010.

[23] P. Gonzalez, A. Griesmayer, and A. Lomuscio. Verifying GSM-based business artifacts. In *International Conference on Web Services (ICWS)*, pages 25–32, 2012.

[24] P. Gonzalez, A. Griesmayer, and A. Lomuscio. Model checking gsm-based multi-agent systems. In *ICSOC*, pages 54–68, 2013.

[25] P. Gonzalez, A. Griesmayer, and A. Lomuscio. Verification of gsm-based artifact-centric systems by predicate abstraction. In *ICSOC*, pages 253–268, 2015.

[26] B. B. Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *PODS*, pages 163–174, 2013.

[27] G. Holzmann. *Spin Model Checker, The: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[28] R. Hull et al. Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events. In *ACM DEBS*, 2011.

[29] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 125–136. ACM, 1972.

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

[31] R. Lazić, T. Newcomb, J. Ouaknine, A. W. Roscoe, and J. Worrell. Nets with tokens which carry data. *Fundamenta Informaticae*, 88(3):251–274, 2008.

[32] Y. Li, A. Deutsch, and V. Vianu. A Spin-based verifier for artifact systems. *arXiv preprint*, arXiv:1705.09427, 2017.

[33] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

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

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

[36] Object Management Group. Case Management Model and Notation (CMMN), 2014.

[37] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[38] P.-A. Reynier and F. Servais. Minimal coverability set for Petri nets: Karp and Miller algorithm with pruning. In *Int'l. Conf. on Application and Theory of Petri Nets and Concurrency*, pages 69–88. Springer, 2011.

[39] R. L. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[40] F. Rosa-Velardo and D. de Frutos-Escrig. Decidability and complexity of Petri nets with unordered data. *Theoretical Computer Science*, 412(34):4439–4451, 2011.

[41] N. Sidorova, C. Stahl, and N. Trčka. Soundness verification for conceptual workflow nets with data: Early detection of

errors with the most precision possible. *Information Systems*, 36(7):1026–1043, 2011.

[42] A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, 1994.

[43] 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.

[44] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[45] W. M. Van Der Aalst. Business process management: a comprehensive survey. *ISRN Software Engineering*, 2013.

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

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

[48] A. H. Watson, D. R. Wallace, and T. J. McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

[49] W.-D. Zhu et al. *Advanced Case Management with IBM Case Manager*. IBM Redbooks, 2015.