Practical Verification of Hierarchical Artifact Systems

Yuliang Li
Co-Advised by Alin Deutsch and Victor Vianu
UC San Diego
La Jolla, California
yul206@eng.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 by studying the implementation of a full-fledged and efficient artifact verifier for a variant of the Hierarchical Artifact System (HAS) model presented in [9]. With a family of specialized optimizations to the classic Karp-Miller algorithm, our verifier performs >10x faster than a nontrivial Spin-based baseline on real-world workflows and is scalable to large synthetic workflows.

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 [14], successfully deployed in practice [3, 5, 18] 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) [7, 11]. 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) [15, 12].

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 was 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. Along this line, decidability and complexity results were shown for different versions of the verification problem with various expressiveness of the artifact models, as reviewed in the next section.

The project described in this paper bridges the gap between the theory and practice of artifact verification by providing the first implementation of a full-fledged and efficient artifact verifier. The artifact model we use is a variant of the *Hierarchical Artifact System* (HAS) model of [9], which captures core elements of IBM's GSM model. Rather than building on top of an existing program verification tool such as Spin, which we have shown to have strong limitations, we implemented our verifier from scratch. The implementation is based on the classic Karp-Miller algorithm [16], with a family of specialized optimizations to boost performance. The experimental results show that our verifier performs an order of magnitude faster compared to a baseline implementation using Spin [10] on specifications based on real-world BPMN workflows [2], and scales well on large synthetic workflows. To the best of our knowledge, our artifact verifier is the first implementation with full support of unbounded data.

Proceedings of the VLDB 2017 PhD Workshop, August 28, 2017. Munich, Germany. Copyright (c) 2017 for this paper by its authors. Copying permitted for private and academic purposes.

2. BACKGROUND AND RELATED WORK

[8, 6] studied the verification problem for a bare-bones variant of artifact systems, in which each artifact consists of a flat tuple of evolving values and the services are specified by simple pre-and-post conditions on the artifact and database. The verification problem was to check statically whether all runs of an artifact system satisfy desirable properties expressed in LTL-FO, an extension of linear-time temporal logic where propositions are interpreted as existential first-order logic sentences on the database and current artifact tuple. In order to deal with the resulting infinite-state system, a symbolic approach was developed in [8] to allow a reduction to finite-state model checking and yielding a PSPACE verification algorithm for the simplest variant of the model (no database dependencies and uninterpreted data domain). In [6] the approach was extended to allow for database dependencies and numeric data testable by arithmetic constraints.

In our previous work [9], we made significant progress on several fronts. We introduced the HAS model, a much richer and more realistic model abstracting the core elements of

the GSM model. The model features task hierarchy, concurrency, and richer artifact data (including updatable artifact relations). In more detail, a HAS consists of a database and a hierarchy (rooted tree) of *tasks*. Each task has associated to it local evolving data consisting of a tuple of artifact variables and an updatable artifact relation. It also has an associated set of *services*. Each application of a service is guarded by a pre-condition on the database and local data and causes an update of the 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 input parameters, and receive back a result if the child task completes. To express properties of HAS we introduce *hierarchical* LTL-FO (HLTL-FO), which is similar to LTL-FO but adapted to the hierarchy. The main results of [9] establish the complexity of checking HLTL-FO properties for various classes of HAS, highlighting the impact of various features on verification.

3. MODEL AND EXAMPLE

The artifact model used in our implementation is a variant of the HAS model of [9] denoted HAS*, which differs from the HAS model used in [9] in two respects. On one hand, it *restricts* HAS by disallowing arithmetic in service pre-and-post conditions, and requires the underlying database schema to use an *acyclic* set of foreign keys, as in the widely used Star (or Snowflake) schemas [17]. On the other hand, HAS* *extends* HAS by allowing an arbitrary number of artifact relations in each task, arbitrary variable propagation, and more flexible interactions between tasks. As shown by our real-life examples, HAS* is powerful enough to model a wide variety of business processes, and so is a good vehicle for studying the implementation of a verifier. Moreover, despite the extensions, the complexity of verifying HLTL-FO properties of HAS* can be shown to remain EXPSPACE, by adapting the techniques of [9].

We illustrate the HAS* model with a simplified example of order fulfillment business process based on a real-world BPMN workflow. The workflow allows customers to place orders and suppliers to process the orders. It has the following database schema:

- `CUSTOMERS`(<u>`id`</u>, `name`, `address`, `record`)
  `ITEMS`(<u>`id`</u>, `item_name`, `price`, `in_stock`)
  `CREDIT_RECORD`(<u>`id`</u>, `status`)

In the schema, the `id`'s are key attributes, and `record` is a foreign key referencing `CREDIT_RECORD`. The `CUSTOMERS` table contains basic customer information and `CREDIT_RECORD` provides each customer's credit rating. The `ITEMS` table contains information on all the items. The artifact system has 4 tasks: T_1:**ProcessOrders**, T_2:**TakeOrder**, T_3:**CheckCredit** and T_4:**ShipItem**, which form the hierarchy in Figure 1.

Intuitively, the root task **ProcessOrders** serves as a global coordinator which manages a pool of orders and the child tasks **TakeOrder**, **CheckCredit** and **ShipItem** implement the 3 processing stages of an order. At each point in a run, **ProcessOrders** nondeterministically picks an order from its pool, triggers one processing stage, and places it back into the pool upon completion.

ProcessOrders: The task has the artifact variables: `cust_id`, `item_id`, `status` which store basic information of an order. It also has an artifact relation `ORDERS`(`cust_id`, `item_id`, `status`) storing the orders to be processed.

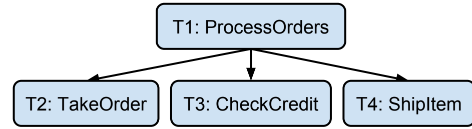

Figure 1: Tasks Hierarchy

The 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 non-deterministically chosen and removed from `ORDERS` for processing, and (`cust_id`, `item_id`, `status`) is set to be the chosen tuple. When *StoreOrder* is called, the current order (`cust_id`, `item_id`, `status`) is inserted into `ORDERS`. The latter two services are specified as follows (the specification consists of a pre-condition, a post-condition, and an update to the `ORDERS` artifact relation):

StoreOrder:
Pre: `cust_id` ≠ `null` ∧ `item_id` ≠ `null` ∧ `status` ≠ "`Failed`"
Post: `cust_id` = `null` ∧ `item_id` = `null` ∧ `status` = "`Init`"
Update: {+`ORDERS`(`cust_id`, `item_id`, `status`)}

RetrieveOrder:
Pre: `cust_id` = `null` ∧ `item_id` = `null` // Post: `True`
Update: {−`ORDERS`(`cust_id`, `item_id`, `status`)}

TakeOrder: 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 initialized to "`OrderPlaced`". The task contains `cust_id`, `record` and `status` as variables and all are return variables to the parent task. There are two services called *EnterCustomer* and *EnterItem*, that allow the customer to enter her and the item's information. The `CUSTOMERS` and `ITEMS` tables are queried to obtain the customer ID and item ID. These two services can be called multiple times to allow the customer to modify previously entered data. The task's termination condition is `cust_id` ≠ `null` ∧ `item_id` ≠ `null`, at which time its variables are returned to its parent task **ProcessOrders**.

CheckCredit: This task checks the financial record of a customer and decides whether the supplier will go ahead with the sale. It is called when `status` = "`OrderPlaced`". It has artifact variables `cust_id` (input variable), `record` and `status`. When the credit record is good, `status` is set to "`Passed`", and otherwise to "`Failed`". After `status` is set, the task terminates and returns `status` to the parent task. The task has a single service *Check* performing the credit check.

Check:
Pre: `true` // Post:
∃n∃a `CUSTOMERS`(`cust_id`, n, a, `record`)∧
(`CREDIT_RECORD`(`record`, "`Good`") → `status` = "Passed")∧
(¬`CREDIT_RECORD`(`record`, "`Good`") → `status` = "Failed")

Note that in a service we can also specify a set of propagated variables whose values stay unchanged when the service is applied. In *Check*, only `cust_id` is a propagated variable and others will be assigned new values.

ShipItem: This task checks whether the desired item is in stock by looking up the `item_id` in the `ITEMS` table to see whether the `in_stock` attribute equals "`Yes`". If so, the item is shipped to the customer (`status` is set to "`Shipped`") otherwise the order fails (`status` is set to "`Failed`"). This task is specified similarly to **CheckCredit** (details omitted).

Properties of HAS* can be specified in LTL-FO. In the

above workflow, we can specify a temporal property saying *"If an order is taken and the ordered item is out of stock, then the item must be restocked before it is shipped."* It can be written in LTL-FO as:

$$\forall i \ \mathbf{G}(\texttt{EnterItem} \wedge \texttt{item_id} = i \wedge \texttt{instock} = \text{"No"}) \rightarrow (\neg(\texttt{ShipItem} \wedge \texttt{item_id} = i) \ \mathbf{U} \ (\texttt{Restock} \wedge \texttt{item_id} = i))$$

4. VERIFIER IMPLEMENTATION

Although decidability of verification was shown in [9], a naive implementation of the EXPSPACE algorithm outlined there would be wholly impractical. Instead, our implementation brings to bear a battery of optimization techniques crucial to performance. This approach of [9] is based on developing a symbolic representation of the runs of a HAS*. In the representation, each snapshot is summarized by:

(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

The heart of the proof in [9] is showing that it is sufficient to verify symbolic runs rather than actual runs. Observe that because of (ii), the symbolic representation is not finite state. Indeed, (ii) requires maintaining a set of counters, which can grow unboundedly. Therefore, the verification algorithm relies on a reduction to state reachability in Vector Addition Systems with States (VASS) [4]. A VASS is a finite-state machine augmented with positive counters that can be incremented and decremented (but not tested for zero). This is essentially equivalent to a Petri Net.

A direct implementation of the above algorithm 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, our implementation 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 an exponentially smaller search space. Then our verifier performs the (repeated) state reachability search using the classic Karp-Miller algorithm [16] with three specialized optimizations to further accelerate the search. We discuss these next.

State Pruning The classic Karp-Miler algorithm is well-known to be inefficient and pruning is a standard way to improve its performance [16]. We introduce a new pruning technique which can be viewed as a generalization of the strategies in [16]. The high-level idea is that when a new state I is found, if there exists a reached state I' such that all states reachable from I are also reachable from I', then we can stop exploring I immediately. In this case, we call I' a *superstate* of I and I a *substate*. Similarly, if there exists a reached state I' which is a substate of I, then we can prune I' and its successors. Compared to [16], our pruning is more aggresive, resulting in a much smaller search space. As shown in Section 5, the performance of the verifier is significantly improved.

Data Structure Support When the above optimization is applied, a frequent operation during the search is to find substates and superstates of a given candidate state in the current set of reached symbolic states. This operation becomes the performance bottleneck when there is a large number of reached states. We accelerate the superstate and substate queries with a Trie index and an Inverted-Lists index, respectively.

Static Analysis The verifier statically analyzes and simplifies the input workflow with a preprocessing step. We notice that in real workflows, some contraints in the specification can never be violated in a symbolic run, and thus can be removed. For example, for a constraint $x = y$ in the specification, where x, y are variables, if $x \neq y$ does not appear anywhere in the specification and is not implied by other constraints, then $x = y$ can be safely removed from the specification without affecting the result of the verification algorithm.

5. EXPERIMENTAL RESULTS

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

The Real Set As the artifact approach is still new to the industry, real-world processes available for evaluation are limited. We therefore built an artifact system benchmark specific to business processes, by rewriting the more widely available process-centric BPMN workflows as HAS* specifications. There are numerous sources of BPMN workflows, including the official BPMN website [2], that provides 36 workflows of non-trivial size. To rewrite these workflows into the HAS*, we manually added the database schema, artifact variables/relations, and services for updating the data. Among the 36 real-world BPMN workflows collected from the official BPMN website `bpmn.org`, our model is sufficiently expressive to specify 32 of them in HAS* and can thus be used for performance evaluation. The remaining ones cannot be expressed in HAS* because they involve computing aggregate functions or updating the artifact relations in ways that are not supported in the current model. We will consider these features in our future work.

The Synthetic Set The second benchmark we used for evaluation is a set of randomly generated HAS specifications. All components of each specification, including DB schema, task hierarchy and services, are generated fully at random of a certain size. The ones with empty search space due to unsatisfiable conditions are removed from the benchmark. Table 1 shows some statistics of the benchmarks.

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

Table 1: Statistics of the Two Sets of Workflows

Baseline and Setup We compare our verifier with a simpler implementation built on top of Spin, a widely used software verification tool [10]. Building such a verifier is by itself a challenging task since Spin is incapable of handling data of unbounded size, present in the HAS* model. We managed to build a Spin-based verifier supporting a restricted version of out model, without updatable artifact relations.

As the read-only database can still have unbounded size and domain, the verifier requires a set of nontrivial translations and optimizations. The details will be discussed in a separate paper.

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 workflow in each dataset, we ran our verifiers to test a randomly generated liveness property. The time limit of each run was set to 10 minutes. For fair comparison, since the Spin-based verifier (Spin-Opt) cannot handle artifact relations, we ran both the full Karp-Miller-based verifier (KM), and the Karp-Miller-based verifier with artifact relations ignored (KM-NoSet).

Performance Table 2 shows the results on both sets of workflows. The Spin-based verifier achieves acceptable performance on the real set with an average elapsed time of few seconds and only 1 timeout instance. However, it failed on most runs (109/120) in the synthetic set of workflows. On the other hand, both KM and KM-NoSet achieve average running times below 1 second and with no timeout on the real set, and the average running time is in seconds on the synthetic set, with only 3 timeouts. The presence of artifact relations introduced only a negligible amount of overhead in the running time. Compared with the Spin-based verifier, the KM-based approach is >10x faster in the average running time and more scalable to large workflows.

Mode	Real		Synthetic	
	Avg(Time)	#Timeout	Avg(Time)	#Timeout
Spin-Opt	3.111s	1	67.01s	109
KM-NoSet	**.2635s**	0	**3.214s**	3
KM	**.2926s**	0	**3.355s**	2

Table 2: Performance of the two Verifiers

Cyclomatic Complexity To better understand the scalability of the Karp-Miller-based approach, we measured the difficulty of verifying each workflow using a metric called the cyclomatic complexity [13], which is widely used in software engineering to measure the complexity of program modules. Figure 2 shows that the elapsed time increases exponentially with the cyclomatic complexity. According to [13], it is recommended that any well-designed program should have cyclomatic complexity at most 15 in order to be readable and testable. Our verifier successfully handled all workflows in both benchmarks with cyclomatic complexity less than or equal to 17, which is above the recommended level. For instances with cyclomatic complexity above 15, our verifier only timed out in 2/24 instances (8.33%).

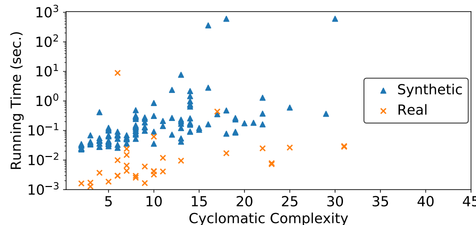

Figure 2: Running Time vs. Cyclomatic Complexity

Comparing Different Optimizations We show next the effect of our 3 optimization techniques: state pruning (SP), static analysis (SA) and data structure support (DSS), by rerunning the experiment with the optimization turned off, and comparing the difference. Table 3 shows the average speedups of each optimization in both datasets. We also present the trimmed averages of the speedups (i.e. removing the top/bottom 5% speedups before averaging) which is less sensitive to extreme values.

Dataset	SP		SA		DSS	
	Mean	Trim.	Mean	Trim.	Mean	Trim.
Real	2943.58x	**55.31x**	1.80x	**1.66x**	1.90x	1.24x
Synthetic	494.57x	**180.82x**	**17.92x**	0.92x	1.45x	**1.27x**

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

Table 3 shows that the effect of state pruning is the most significant in both sets of workflows, with an average (trimmed) speedup of 55x and 180x in the real and synthetic set, respectively. The static analysis optimization is more effective in the real set (1.6x improvement) but its effect in the synthetic set is less obvious. It creates a small amount (8%) 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. Finally, the data-structure support provides a consistent ∼1.2x average speedup in both datasets.

Acknowledgement This work was supported in part by the National Science Foundation under award IIS-1422375.

6. REFERENCES

[1] MIST - a safety checker for petri nets and extensions. `https://github.com/pierreganty/mist/wiki`.

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

[3] 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.

[4] 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.

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

[6] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. TODS, 37(3):22, 2012.

[7] 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.

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

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

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

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

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

[13] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

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

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

[16] P.-A. Reynier and F. Servais. Minimal coverability set for petri nets: Karp and miller algorithm with pruning. In *International Conference on Application and Theory of Petri Nets and Concurrency*, pages 69–88. Springer, 2011.

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

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