Abstract

Business Artifacts are the core entities used by businesses to record information pertinent to their operations. Business operational models are representations of the processing of business artifacts. Traditional process modeling approaches focus on the actions taken to achieve a certain goal (verb-centric). Business artifact-centric modeling starts by identifying what is acted upon (noun-centric), and constructs business operational models by identifying the tasks/actions that business actors execute to add business value. In this paper, we identify important classes of properties on artifact-centric operational models. In particular, we focus on persistence, uniqueness and arrival properties. To enable a static analysis of these properties, we propose a formal model for artifact-centric operational models. We focus on persistence, uniqueness and arrival properties. We show that the formal model guarantees persistence and uniqueness. We prove that while checking an arrival property is undecidable in general, under a restricted version of the formalism, an arrival property can be checked in \text{EXPTIME}.

1 Introduction

Any business needs to maintain records of information pertinent to their operations. Information about the key aspects of a business operation is used to evaluate the effectiveness in producing the business products. In recent years regulatory measures require significant rethinking about the level of traceability to ensure accountability of business actions that meet compliance standards. Businesses of today, small or large, maintain such records, sometimes surprisingly still in paper format, sometimes in more electronic formats such as spreadsheets and sometimes within larger management systems such as ERP solutions.

In our day-to-day experience one can observe many uniquely identifiable entities that are used by businesses to capture information pertinent to a business operation. For example, consider a standard banking process where a customer withdraws money from his/her banking account. The customer fills out a withdrawal slip which captures the customer account number, name, date and the amount s/he wants to withdraw. A teller receives this request document and matches it against the customers account status and properly fulfills the request by withdrawing the requested amount from the customers account. The withdrawal slip is then filed, eventually scanned and added to the customers records. Another common example is the management of orders in a restaurant. The key entity here is the guest check, which captures all orders, the price for each order, the total, when it has been paid and what the means of payment were. The restaurant thereby records all important information on the guest check to account for all orders placed and money received from the customers, which is matched against the actual cash receipts in the register. The Withdrawal Slip and the Guest Check are the core entities used to manage business operations for the bank or the restaurant respectively.

We call these core entities Business Artifacts. In recent years several teams within IBM Research have conducted over a dozen studies with a variety of enterprises from different industry vertical. The core aspect of these studies was to identify business artifacts and how they are used to manage business operations. A business artifact is an identifiable, self-describing unit-of information through which business stakeholders add value to the business. A Withdrawal Slip, a Receipt, a Purchase Order, a Service Order, etc. are typical examples of business artifacts. A purchase order is a unique entity with well-defined business intent, namely capturing the order of a customer and record-
Traditional business process modeling techniques focus on the actions taken to achieve a certain goal (often referred to as verb-centric). Hence, business stakeholders describe their business by stating what they do, e.g., purchase order, insurance claim, and while doing C we also do D. Our approach proposes to focus on what is acted upon, thus describing business operations by first identifying the things that matter to their business processes. Second, how these things are processed to achieve a certain goal. Modeling business operations using business artifacts is thus a noun-centric approach.

In our client engagements we have made several observations with respect to the value of artifact-centric operational modeling. First, consolidating business processes in enterprises of today is an arduous task. Large enterprises are often grown out of mergers and acquisitions and thus many business processes exist in various different forms within the same enterprise. Consolidation efforts are often stalling the unification process as business process owners cannot agree on the one-for-all process. The reluctance of business stakeholders to agree on one process is somewhat a cultural problem, yet we have found that introducing the noun-centric artifact approach helps business stakeholders more easily to agree on the business operations. We empirically observed that it is easier for business stakeholders to agree on the things worked upon to managing their business rather than on how they want to establish a business process. Agreeing on the artifacts that capture key information pertinent to the business context is the basis for asking how these artifacts are being processed to achieve the business goal. Second, over the last few years a research team at IBM Research has developed a toolkit containing tools, technologies and methods to use the notion of business artifacts and build SOA solutions by maintaining the abstraction in terms of artifacts from operational modeling down to solution design to implementation. This has been done in the context of Model-driven Business Transformation and applied in various real-world engagements.

In this study we investigate a formalization of artifact-centric modeling of business operations (also referred to as business operational modeling as opposed to business process modeling) and study some static analysis questions.

The main technical contributions of this paper are:

- We investigate business artifact-centric operational modeling approach of [21], and for each construct we propose an appropriate formalism that enables automated analysis of such models.

- We identify some important classes of properties such as persistence, uniqueness, arrival, encounter, semi-synchronizability, redundancy, access independence, data independence. We focus on persistence, uniqueness, and arrival.

- We show that the proposed formalism guarantees persistence and uniqueness. We prove that checking an arrival property is undecidable in general. Under a restricted version of the formalism, we demonstrate that an arrival property can be checked in EXPTIME.

The rest of the paper is organized as follows: In Section 2 we give an informal introduction to artifact-centric operational modeling with an example. In Section 3 we describe a set of important properties one may want to check against operational models. In Section 4 we present our formalism. In Section 5 we explain our results. The conclusion and future work are provided at the end.

1.1 Related Work

Many tools and techniques have been used for the development of business process models over the last two decades [13, 18, 26, 16]. These approaches have followed a process-centric approach focused on the control and coordination of tasks [9]. The importance of a data-centric view of processes is advocated in [2] and [11]. In [2], the authors encourage an “object view” of scientific workflows where the data generated and used is the main focus. In [11], the author investigates “attribute-centric” workflows where attributes and modules have states. [17] proposes a mixed approach which can express both control and data flow. Another mixed approach is proposed by BPMN [4]. Compared to these approaches, our work also follows a data-centric view, but the granularity of our notion of data is different.

Our work is also related to object-oriented programming formalisms that have been proposed in [1] and [12]. [1] proposes a model for object-oriented programming and studies consistency of method calls with respect to method signatures. Our method formalism is syntactically similar to the one of [12] where an arrival-like question was studied. In our case, however, the methods are nonrecursive and the arrival is formulated between configurations instead of methods.

Other related work includes object histories of [10]. Inflow schemas of [14], the scripts of [19], and object migrations of [25]. [10] defines a mathematical formalism that captures the main aspects of the historical data for objects. Our notion of a run is similar to an “object history” of [10], but in our case a computation-tuple sequence also includes states of tasks. The computation model for tasks can be viewed as a proper subset of behavior models such as the scripts of TAXIS [19]. Inflo flow [14] proposes a data-centric modeling but a computation model is not provided for
tasks. [25] investigates the formal characterization of object migration patterns under different update languages.

Our intention to use finite state machines for business artifacts resembles the typestate concept proposed in [24]. Typestate is a refinement of a type and a machine over typestates is used to define syntactically well-formed but semantically undefined sequences of operations on variables of a type. In this work, we do not provide an extensive language to describe constraints on the artifacts and their relationships; however, the results on the constraint languages for object models such as [22] and [6] can give some hints along this direction.

There have been studies also on static analysis of e-business process specifications. In [20] and [23], the authors analyze BPEL specifications via converting them to Petri Nets. In [8], the authors present a set of tools and techniques for analyzing interactions of composite asynchronous BPEL services. [15] provides a verification tool for web service orchestration, process algebra as the underlying formalism. In [5], the authors use a mix of techniques from logic and automated verification, and propose a tool to verify data-driven Web services. In [7], specifications are modeled in UML, in the form of Message Sequence Charts (MSCs), and transformed into the Finite State Process notation (FSP) to concisely describe and reason about the concurrent executions.

2 Business Artifact-centric Operational Modeling

In this section, we briefly describe the terminology and the constructs in artifact-centric operational modeling. Artifact-centric models consist of 3 key constructs: Business artifacts, business tasks, and repositories [21].

Figure 1. Graphical Notations of Modeling Primitives

A business artifact is an identifiable, self-describing unit-of-information. Business stakeholders add value to the business through business artifacts. For instance, in a restaurant, guest checks are the key business artifacts. A guest check captures what a customer orders and the status of these orders as they are processed. Business stakeholders such as waiters and cashiers update the check in the course of a dining experience. A guest check thereby has a lifecycle from the creation when the customer arrives to the archival when the customer leaves.

A business task (or simply task) describes the work acting upon an artifact by which a business role adds measurable business value to this artifact. We require a task to generate business value and hence, update an artifact. This condition helps in defining the granularity of a task or the task boundaries. Imagine a simple scenario where two tasks, T1 and T2 work on an artifact consisting of ten name-value pairs. A business stakeholder could determine that completion of T1 will require an update of, say, attributes 2-5 of the artifact and T2 requires an update of attributes 6-10. Therefore, adding business value in this case can be clearly defined by the business stakeholder who thereby determines the boundaries of a task.

A repository describes a waiting shelf or a buffer for an artifact. Tasks can push an artifact into a repository and pull an artifact out of a repository.

Example[1] Figure 2 illustrates an operational model of a restaurant. This restaurant uses four artifact types while conducting its business: “guest checks”, “receipts”, “kitchen orders”, and “cash balance”.

We can informally describe the operations of this restaurant as follows: When a customer walks in the restaurant, the customer is greeted and a guest check artifact is created with the customer information such as the customer name and table number. It is then placed into the repository for “open” guest checks (Open GCs). When the customer gives an order, “AddItem” task is triggered. This task retrieves the customer’s guest check, creates a kitchen order and adds it into the “pending kitchen orders” repository. After the kitchen order is prepared, if it passes the quality test, it is inserted into the “ready kitchen orders” repository. Otherwise, it is sent back to the pending kitchen orders repository to be re-prepared. The required quality by the quality test depends on the customer. If the customer is a “special” customer then the order must have a high qual-

Figure 2. Guest Check Operation Model of Restaurant(GC=GuestCheck, RC=Receipt, KO=KitchenOrder, CB=CashBalance)
ity, otherwise, a regular quality is “ok”. The kitchen orders passing the quality test is delivered, and the delivery time of the kitchen order is recorded on the guest check and the kitchen order artifact is “archived”.

When the check is requested, a receipt is prepared and the guest check is “closed”. If the customer disagrees on the receipt amount, the receipt is recalculated. When the payment of the receipt is successfully done, the receipt is added to the cash balance of the restaurant and then, the receipt and the guest check are archived.

3 Static Analysis of Operational Models

Artifact-centric operational models are specifications of businesses at an abstract level and they are used as the starting point to obtain implementations. Therefore, the “correctness” of models is of critical importance to businesses. By correctness, we mean models should satisfy “desired” properties derived from the business logic and context. Towards this direction, we identified several classes of properties. These properties help to understand the discrepancies between what is desired and what a model satisfies.

Arrival: Can an artifact A arrive into a repository R? In our example, a guest check should be able to arrive into the archived-guest-checks repository. This shows the completion of the guest check processing is possible.

Persistence: Once an artifact is created, does it persist or can it disappear? In order for a business to operate, the “important” business data should never disappear and should always be locatable. For example, a guest check should never be “destroyed” or get “lost”, because it contains key information pertinent to the operations of the restaurant.

Uniqueness: Can an artifact appear in more than one “places” at once? An “unambiguous” specification of business operations require “important” business data to be in one business state at any time. For instance, a receipt shouldn’t be in both paid-receipts and archived-guest-checks repositories at the same time.

Encounter: Can two different artifacts encounter? For instance, in our example, a receipt and a kitchen order never encounter, while a receipt and a guest check may encounter in update-cash-balance task.

Semi-synchronizability: An artifact’s “progress” can be dependent on the existence of another artifact. For instance, the archival of a kitchen order requires the existence of the guest check; therefore, the kitchen order is semi-synchronizable with the guest check. On the other hand, the guest check can be moved to closed-guest-checks repository without the arrival of the kitchen order. Therefore, the guest check is not semi-synchronizable with the kitchen order.

Task/Repository Redundancy: Is a task (or repository) redundant? (The removal of it doesn’t “impact” the “behavior” of the model). An operational model can be optimized by eliminating the redundancies.

Access Independence of Tasks: Do two tasks access different parts of the same artifact? For instance, the tasks prepare-receipt and recalculate-receipt are not access independent, because they access the receipt amount.

Data Independence: Do two artifacts have a data relationship? For instance, the receipt and the cash balance are not data independent because the cash balance amount depends on the receipt amount. On the other hand, the guest check and the cash balance have no data relationship; therefore, they are data independent.

In this paper, we focus on arrival, persistence, and uniqueness properties. Automated analysis of these properties require a formalization for operational models. To achieve this, in the next section, we propose a formal model.

4 A Formal Model For Business Operations

In this section, we propose a formalism for modeling artifact-centric business operations and the key concepts including “artifacts” and “operational models”.

We assume two pairwise disjoint domains: an infinite set of uninterpreted elements denoted by dom and on which only the equality relation is defined, and an infinite densely ordered set dom<. The values in dom = dom ∪ dom< are called constants. We assume an infinite set of business artifact instance identifiers denoted by domid on which only the equality relation is defined. We also assume an infinite set NA of business artifact type names.

We define three kinds of atomic types: dle, deq, and a business artifact type to which a unique name in NA is associated (more explanation comes later). We define a mapping dom between these types and the domains such that dom(deq) = dom<, dom(dle) = dom, dom(τ) = domid where τ ∈ NA.

We also define a data function type. A data function type has the form S1, . . . , St → T1, . . . , Tm where Si’s, Ti’s are atomic types. An instance f of such a type is a mapping f : S1 × · · · × St → T1 × · · · × Tm with a finite domain such that f maps a tuple (a1, . . . , at) ∈ dom(S1)× · · · × dom(Si) to a tuple (b1, . . . , bm) ∈ dom(T1)× · · · × dom(Tm).

Let f be an element in dom<. As a shorthand we define a set type f(S1, . . . , St) as the data function type f : S1, . . . , St → dle such that we say an instance of f contains an element (a1, . . . , at) iff f(a1, . . . , at) = 1.

An artifact comparison over x, y is of the form x = y where x, y are variables of the same artifact type. Scalar comparisons over x, y, c are of the forms x ≥ y and x < c where x, y are variables of the type dle (resp. deq), c is a constant in dom< (resp. dom), θ ∈ {=, ≠} (resp. θ ∈ {<, ≤, >, ≥, =, ≠}).

4.1 Business Artifact

We now present the central notion of an artifact. A business artifact contains information pertinent to the business
context at hand and business operations represent the lifecycle of artifacts from creation to completion. Information is represented using attribute values. Furthermore a set of methods and a state machine describe the act of processing on the artifact from the artifact’s perspective. Our intention to use state machines is to define semantically defined sequences of operations on artifacts similar to the typestate concept of [2,4].

Although the notion except for the state machine resembles a class definition in object oriented models, our key focus is on the life cycles of artifacts or “process organization” rather than the ease of programming or data organization.

**Definition:** A *business artifact type* (shortly an artifact type) is a tuple \((L, P, M)\) such that

- \(L\) is a set of attribute names with associated types.
- \(P\) is a set of methods with distinct names.
- \(M\) is a (possibly nondeterministic) finite state machine and its transitions are labeled with method names from \(P\) such that \(M\) recognizes the language of the allowed sequences of method invocations.

A *method* has four components:

1. *name*,
2. *input parameters*: \(u_1, \ldots, u_m\) with atomic types \(S_1, \ldots, S_m\),
3. *output parameters*: \(w_1, \ldots, w_n\) with atomic types \(T_1, \ldots, T_n\),
4. *body*: A sequence of statements each is in one of the following forms:
   - \(x := y\) where \(x\) is an output parameter or in \(L\), \(y\) is an input parameter or in \(L\), and \(\text{typeOf}(x) = \text{typeOf}(y)\),
   - \(x \cup = (u_{i_1}, \ldots, u_{i_j} \rightarrow u_{k_1}, \ldots, u_{k_l})\) where \(x \in L\) and \(\text{typeOf}(x) = S_{i_1}, \ldots, S_{i_j} \rightarrow S_{k_1}, \ldots, S_{k_l}\), and \(i_x, k_y \in \{1, \ldots, m\}\),
   - \(x := (u_{k_1}, \ldots, u_{k_l}) := x(u_{i_1}, \ldots, u_{i_j})\) where \(x \in L\) and \(\text{typeOf}(x) = S_{i_1}, \ldots, S_{i_j} \rightarrow T_{k_1}, \ldots, T_{k_l}\) and \(i_x \in \{1, \ldots, m\}, k_y \in \{1, \ldots, n\}\),
   - \(w_i := (u_{i_1}, \ldots, u_{i_j} \rightarrow u_{k_1}, \ldots, u_{k_l}) \in x\) where \(x \in L\) and \(\text{typeOf}(x) = S_{i_1}, \ldots, S_{i_j} \rightarrow S_{k_1}, \ldots, S_{k_l}\), and \(i_x, k_y \in \{1, \ldots, m\}\), and \(T_i\) is defined.

An artifact instance of an artifact type \((L, P, M)\) is a pair \((id, \lambda)\) where \(id \in \text{dom}(\text{id})\), and \(\lambda\) is a partial mapping such that \(\lambda(\text{state})\) is in \(\text{statesOf}(M)\), and for every \(x \in L\), if defined, \(\lambda(x)\) is in \(\text{dom}(\text{typeOf}(x))\). Let \(x.id\) represent the id of an artifact instance. We say a set of artifact instances is consistent, if for every two different \(x, y\) in the set, \(x.id \neq y.id\).

Let \(\mathcal{A}\) represent a set of artifact types with the following property: For every type \(\tau\) in \(\mathcal{A}\), any type referenced by \(\tau\) is also in \(\mathcal{A}\). Let \(\mathcal{A}\) represent the set of all consistent sets of artifact instances.

### 4.1.1 Method Semantics

Let a *valuation of the input and output parameters of a method* be a partial mapping \(\beta\) on input and output parameters of the method such that for every parameter \(x, \beta(x)\) is, if defined, in \(\text{dom}(\text{typeOf}(x))\). Let \((id, \lambda), (id, \lambda')\) be two artifact instances of an artifact type \((L, P, M)\), \(m\) be a method in \(P\) having input parameters \(u_1, \ldots, u_m\) with atomic types \(S_1, \ldots, S_m\) and output parameters \(w_1, \ldots, w_n\) with atomic types \(T_1, \ldots, T_n\). Let \(\beta, \beta'\) be two valuations of input and output parameters of \(m\).

For a statement \(s\) of \(m\), we say \((\lambda', \beta')\) can derived from \((\lambda, \beta)\) with \(s\), denoted by \((\lambda, \beta) \xrightarrow{s} (\lambda', \beta')\), if \(\lambda'\) and \(\beta'\) can be derived from \(\lambda\) and \(\beta\) according to the following rules:

- If \(s\) has the form \(x := y\), and, \(y\) is an input parameter and \(\beta(y)\) is defined, or \(y\) is in \(L\) and \(\lambda(y)\) is defined, then \(\lambda'\) and \(\beta'\) are derived as follows:
  - \(\lambda'(x) = \lambda(y)\) if \(x, y \in L\).
  - \(\lambda'(x) = \beta(y)\) if \(x \in L\) and \(y\) is an input parameter.
  - \(\beta'(x) = \lambda(y)\) if \(x\) is an output parameter and \(y \in L\).
  - \(\beta'(x) = \beta(y)\) if \(x\) is an output parameter and \(y\) is an input parameter.
- For every \(z \in L - \{x\}\), \(\lambda'(z) = \lambda(z)\) if \(\lambda(z)\) is defined; otherwise, \(\lambda'(z)\) is not defined.
- For every \(z \in \{u_1, \ldots, u_m, w_1, \ldots, w_n\} - \{x\}\), \(\beta'(z) = \beta(z)\) if \(\beta(z)\) is defined; otherwise, \(\beta'(z)\) is not defined.
- \(\lambda'(\text{state}) = q\) for some \(q \in \text{statesOf}(M)\) such that \(\lambda(\text{state})\) is a transition from \(\lambda(\text{state})\) to \(q\) in \(M\).

- If \(s\) has the form \(x \cup = (u_{i_1}, \ldots, u_{i_j} \rightarrow u_{k_1}, \ldots, u_{k_l})\) and \(\beta(z)\) is defined for every \(z \in \{u_{i_1}, \ldots, u_{i_j}, u_{k_1}, \ldots, u_{k_l}\}\), then \(\lambda'\) and \(\beta'\) are derived as follows:
  - \(\lambda(x)\) is not defined, \(\lambda'(x) = f\) for some data function instance \(f\) of type \(S_{i_1}, \ldots, S_{i_j} \rightarrow T_{k_1}, \ldots, T_{k_l}\), where \(f\) is defined as follows:
    - \((i)\) \(f(\beta(u_{i_1}), \ldots, \beta(u_{i_j})) = (\beta(u_{k_1}), \ldots, \beta(u_{k_l}))\);
    - \((ii)\) for every tuple \((a_1, \ldots, a_j)\) different than \((\beta(u_{i_1}), \ldots, \beta(u_{i_j}))\), \(f(a_1, \ldots, a_j)\) is not defined.
  - \(\lambda(x)\) is defined, then \(\lambda'(x) = f'\), for some data function instance \(f'\) of type \(S_{i_1}, \ldots, S_{i_j} \rightarrow T_{k_1}, \ldots, T_{k_l}\) where \(f'\) is defined as follows:
    - \((i)\) \(f'((\beta(u_{i_1}), \ldots, \beta(u_{i_j})) = (\beta(u_{i_1}), \ldots, \beta(u_{i_j}))\);
    - \((ii)\) for every tuple \((a_1, \ldots, a_j)\) different than \((\beta(u_{i_1}), \ldots, \beta(u_{i_j}))\), if \(f(a_1, \ldots, a_j)\) is defined where \(f = \lambda(x)\), then \(f'(a_1, \ldots, a_j)\) =
\( f(a_1, \ldots, a_j); \) otherwise, \( f'(a_1, \ldots, a_j) \) is not defined.

- For every \( y \in L - \{x\} \), \( \lambda'(y) = \lambda(y) \) if \( \lambda(y) \) is defined; otherwise, \( \lambda'(y) \) is not defined.

- For every \( y \in \{u_1, \ldots, u_m, w_1, \ldots, w_n\} \), \( \beta'(y) = \beta(y) \) if \( \beta(y) \) is defined; otherwise, \( \beta'(y) \) is not defined.

- \( \lambda'(\text{state}) = q \) for some \( q \in \text{statesOf}(M) \) such that \( (\lambda(\text{state}), m, q) \in \text{transitionRelationOf}(M) \).

- If \( s \) has the form \( (w_{k_1}, \ldots, w_{k_i}) := (u_{i_1}, \ldots, u_{i_j}) \), then \( \lambda'(z) = \lambda(x) \) if \( \lambda(x) \) is defined; otherwise, \( \lambda'(z) \) is not defined.

- For every \( y \in L \), \( \lambda'(y) = \lambda(y) \) if \( \lambda(y) \) is defined; otherwise, \( \lambda'(y) \) is not defined.

- For every \( y \in \{u_1, \ldots, u_m, w_1, \ldots, w_n\} \), \( \beta'(y) = \beta(y) \) if \( \beta(y) \) is defined; otherwise, \( \beta'(y) \) is not defined.

- \( \lambda'(\text{state}) = q \) for some \( q \in \text{statesOf}(M) \) such that \( (\lambda(\text{state}), m, q) \in \text{transitionRelationOf}(M) \).

We assume a set \( \mathcal{R} \) of repositories each has an associated artifact type in \( \mathcal{R} \).

**Definition:** The following operations are defined on a repository \( R \) (The semantics are defined later.):

- **checkIn:** adds an artifact in \( R \),
- **checkAnyOut:** retrieves an artifact from \( R \),
- **checkOut:** retrieves a specific artifact from \( R \),
- **contains:** checks if \( R \) contains a specific artifact,
- **nonempty:** checks if \( R \) contains any artifact.

Let \( \text{related}(A, \mathcal{R}) \) be true if \( A \subseteq \mathcal{R}, \mathcal{R} \subseteq \mathcal{R} \), the associated type of every \( R \in \mathcal{R} \) is in \( A \), and for every type \( \tau \) in \( A \), there is at least one \( R \in \mathcal{R} \) such that its associated type is \( \tau \).

In the remainder of the paper, we use \( A, \mathcal{R}, \Lambda \) so that \( \text{related}(A, \mathcal{R}) \) is true and \( \Lambda \) is a consistent set of artifact instances each is of a type in \( A \).

A repository mapping \( \gamma \) over \( \mathcal{R} \) and \( \Lambda \) is a mapping \( \gamma : \mathcal{R} \rightarrow 2^\Lambda \) with the following properties: for every repository \( \mathcal{R} \in \mathcal{R} \), \( \gamma(\mathcal{R}) \subseteq \Lambda \), and for all \( x \in \gamma(\mathcal{R}) \), \( x \)'s type is the same with \( \mathcal{R} \)'s associated type, and for all different \( \mathcal{R}_1, \mathcal{R}_2 \in \mathcal{R} \), for all \( y \in \gamma(\mathcal{R}_1) \), for all \( z \in \gamma(\mathcal{R}_2) \), \( y.id \neq z.id \) if \( \gamma(\mathcal{R}_1) \neq \gamma(\mathcal{R}_2) \).

Tasks are functional units of work carried out on artifacts. Tasks retrieve artifacts from repositories, read/update artifact data via invoking the artifact methods, and inserts artifacts into repositories.

**Definition:** A task over \( A, \mathcal{R} \) is a tuple \((L, M)\) where

- \( L \) is a set of variables each is of an artifact type in \( A \), or of the type `Deq` or `Dle`,
- \( M = (\Sigma, S, s_0, S_f, T, l) \) is a deterministic finite state automata where:
  - \( \Sigma \) is a set of statements where a statement has one of the following forms:
    - \( A.\text{new}(x) \)
    - \( R.\text{checkAnyOut}(x) \)
    - \( R.\text{checkOut}(x) \)
    - \( R.\text{checkIn}(x) \)
    - \( \text{reset}(z_1, \ldots, z_i) \)
    - \( x.m(y_1, \ldots, y_k \mapsto z_1, \ldots, z_i) \)
  - \( x \in L \) and \( \text{typeOf}(x) = \tau \) for some artifact type \( \tau \in A; R \in \mathcal{R} \) and \( R \)'s associated type is \( \tau; m \) is a method defined by \( \tau \); each \( y_i \) is in \( L \) or a constant; each \( z_i \) is in \( L \),
  - \( S \) is a set of states, \( S_f \subseteq S \) is a set of final states,
  - \( s_0 \in S_f \) denotes the “initial” state,
  - \( T \), the transition relation, is a subset of \((S \times (\Sigma - \{\text{reset}\}) \times S) \cup (S_f \times \{\text{reset}\}) \times \{s_1\}) \cup ((s_0) \times \{\text{reset}\} \times \{s_1\}) \) for some \( s_1 \in S - \{s_0\} \),
- \( l : T \rightarrow G \) is a labeling function where \( G \) is a set of guards,

4.2 Repository, Task, Operational Model

Repositories are containers for artifacts where they await future processing. Each repository can keep many artifacts of a single artifact type. A repository provides operations for tasks to access the artifacts it contains.
A guard is defined inductively as follows: true is a guard; for every \( x, y \in L \) and a constant \( c \), every artifact comparison over \( x, y \) and every scalar comparison over \( x, y, c \) are guards; \( R \).contains \((x) \) and \( R \).nonempty are guards where \( x \in L \) and \( \text{typeOf}(x) \in \Lambda \), \( R \in \mathbb{R} \) and \( R \)'s associated type equals to \( \text{typeOf}(x) \); \( g_1 \land g_2 \) is a guard where \( g_1, g_2 \) are guards.

Example: Figure 3 illustrates our guarded finite state machine representation for the task AddItem. The task is triggered with a “reset” action which resets all the variables in the context of a task. This action also models the input from the external world; therefore, “gc”, “itemNo”, and “quantity” are set. If the customer is a special customer, if \( \text{type} \neq \text{"special"} \), \( \text{itemNo} \) is set. Finally, \( \text{quantity} \) is set. Otherwise, it is set to have a regular quality (“rq”).

\[
\begin{align*}
&\text{reset(gc, item, quantity)} \\
&R1.\text{checkOut(gc)} \\
&\text{KitchenOrder.new(ko)} \\
&\text{gc.customerType(-type)} \\
&\text{gc.add(ko-)} \\
&R1.\text{checkIn(gc)} \\
&R2.\text{checkIn(ko)} \\
&... \\
&\text{quantity, "hq", gc-) } \\
&[\text{type } \neq \text{"special"}] \\
&\text{ko.setItem(item, quantity, "rq", gc-)} \\
&\text{R2.checkIn(g2-)} \\
&\text{R1.checkIn(gc-)} \\
\end{align*}
\]

\[\text{Figure 3. AddItem Task}\]

Let \( \Pi \) be a set of task instances. A task mapping \( \phi \) over \( \Pi \) and \( \Lambda \) is a mapping \( \phi : \Pi \rightarrow 2^{\Lambda} \) with the following property: for every task instance \( \pi \in \Pi \) of a task \( (L, M) \),
\[
\phi(\pi) \subseteq \{ x \mid x \in \Lambda \text{ and } x.id = \pi(y) \text{ for some } y \in L \}.
\]

We now proceed to define operational models that describe the operations of a business.

**Definition:** An operational model is a tuple \((A, R, T)\) such that \(\text{related}(A, R)\), and \(T\) is a set of tasks over \(A\) and \(R\).

A configuration of an operational model \((A, R, T)\) is a tuple \((\Lambda, \Pi, \phi, \gamma)\) where \(\Lambda \subseteq \Xi\) is a consistent set of artifact instances each is of a type in \(\Lambda\); \(\Pi\) is a set of task instances of tasks in \(T\) with respect to \(\Lambda\); \(\phi\) is a task mapping over \(\Pi\) and \(\Lambda\); \(\gamma\) is a repository mapping over \(R\) and \(\Lambda\) such that \(i)\) for every task in \(T\), there is exactly one task instance in \(\Pi\); \(ii)\) for all \(R \in \mathbb{R}\), for all \(\pi \in \Pi\), for all \(y \in \gamma(R)\), for all \(y \in \gamma(\pi), x.id \neq y.id\); \(iii)\) \(\Lambda = \{ x \mid x \in \phi(\pi) \text{ for } \pi \in \Pi \} \cup \{ x \mid x \in \gamma(\pi) \text{ for } \pi, R \in \mathbb{R} \} \). Intuitively, \(\phi\) refers to the artifact instances that are checked out by the task instances and \(\gamma\) refers to the artifact instances that are located in the repositories.

### 4.3 Computation Semantics

In this section we sketch the semantics of the proposed formalism.

Let \(C = (\Lambda, \Pi, \phi, \gamma)\), \(C' = (\Lambda', \Pi', \phi', \gamma')\) be two configurations. Let \(\pi \in \Pi\) be a task instance of a task \((L, M)\), \(s\) be a statement such that \((\pi(\text{state}), s, q) \in \text{transitionRelationOf}(M)\) for some \(q \in \text{statesOf}(M)\), and \(g\) be the guard assigned to \(s\) in \(M\). We say \(C'\) can be derived from \(C\) with \(s\) of \(\pi\), denoted by \(C \xrightarrow{\pi.g, s, q} C'\), if \(\Lambda', \Pi', \phi', \gamma'\) can be derived from \(\Lambda, \Pi, \phi, \gamma\) according to the rules below:

- If \(s\) has the form \(A.new(x)\), and \(\pi(x)\) satisfies \(g\), and \(\pi(x)\) is not defined, then \(\Lambda', \Pi', \phi', \gamma'\) are derived as follows:
  - Let \((t, \lambda)\) be an artifact instance of the artifact type \(A = (L', P', M')\) such that \(t \in \text{dom}_id - \{ y.id \mid y \in \Lambda \}\), \(\lambda(\text{state}) = \text{initialStateOf}(M')\), and for every \(z \in L'\), \(\lambda(z)\) is not defined.
  - \(\Lambda' = \Lambda \cup \{(t, \lambda)\}\).
  - Let \(\pi'\) be a task instance of the task \((L, M)\) such that \(\pi'(\text{state}) = q, \pi'(x) = t\), and for every \(y \in L - \{x\}, \pi'(y)\) is not defined if \(\pi(y)\) is not defined, and \(\pi'(y) = \pi(y)\), otherwise.
  - \(\Pi' = \Pi - \{\pi\} \cup \{\pi'\}\).
  - For every \(y \in \mathbb{R}, \gamma'(y) = \gamma(y)\).
  - For every \(y \in L' - \{t\}, \phi'(y) = \phi(y)\), and \(\phi'(\pi') = \phi(\pi) \cup \{(t, \lambda)\}\).

- If \(s\) has the form \(R.checkAnyOut(x)\), and \(\pi(x)\) satisfies \(g\), and \(\pi(x)\) is not defined, then \(\Lambda', \Pi', \phi', \gamma'\) are derived as follows:
\[\pi(x) = t\] for some \((t, \lambda) \in \gamma(R)\), then \(\Lambda', \Pi', \phi', \gamma'\) are derived as follows:

- Let \(\pi'\) be a task instance of the task \((L, M)\) such that 
  \[\pi'(state) = q,\] for every \(y \in L\), \(\pi'(y)\) is not defined if \(\pi(y)\) is not defined, and \(\pi'(y) = \pi(y)\), otherwise.
  \[\gamma'(R) = \gamma(R) - \{(t, \lambda)\},\] and for every \(y \in R - \{R\}\),
  \[\gamma'(y) = \gamma(y).\]
  \[\Pi' = \Pi - \{\pi\} \cup \{\pi'.\}\]
  \[\phi'(y) = \phi(y),\] and \(\phi'(\pi') = \phi(\pi) \cup \{(t, \lambda)\}.\]

- If \(s\) has the form \(\mathcal{R}.\text{checkOut}(x)\), and \(\pi\) satisfies \(g, \pi(x)\) is defined, and \(\pi(x) = t\) for some \((t, \lambda) \in \gamma(R)\), then \(\Lambda', \Pi', \phi', \gamma'\) are derived as follows:
  \[\Pi' = \Pi - \{\pi\} \cup \{\pi'.\}\]
  \[\gamma'(R) = \gamma(R) - \{(t, \lambda)\},\] and for every \(y \in R - \{R\}\),
  \[\gamma'(y) = \gamma(y).\]
  \[\Pi' = \Pi - \{\pi\} \cup \{\pi'.\}\]
  \[\phi'(y) = \phi(y),\] and \(\phi'(\pi') = \phi(\pi) \cup \{(t, \lambda)\}.\]

- If \(s\) has the form \(\mathcal{R}.\text{checkIn}(x)\), \(\pi\) satisfies \(g, \pi(x)\) is defined, and \(\pi(x) = t\) for some \((t, \lambda) \in \gamma(R)\), then \(\Lambda', \Pi', \phi', \gamma'\) are derived as follows:
  \[\Pi' = \Pi - \{\pi\} \cup \{\pi'.\}\]
  \[\gamma'(R) = \gamma(R) - \{(t, \lambda)\},\] and for every \(y \in R - \{R\}\),
  \[\gamma'(y) = \gamma(y).\]
  \[\Pi' = \Pi - \{\pi\} \cup \{\pi'.\}\]
  \[\phi'(y) = \phi(y),\] and \(\phi'(\pi') = \phi(\pi) \cup \{(t, \lambda)\}.\]

- A configuration \(C = (\Lambda, \Pi, \phi, \gamma)\) is a root configuration if for every \(\pi \in \Pi\), the following properties are true:
  i) Let \((L, M)\) be the task of which \(\pi\) is an instance. Then, \(\pi(y)\) is undefined for all \(y \in L\). ii) \(\pi(state) = initialStateOf(M)\).
  iii) \(\phi(\pi)\) is empty. The third property implies all artifact instances are in the repositories (i.e., \(\Lambda = \{x \mid x \in \gamma(R)\}\) and \(R \in R\)), and \(\{x \mid x \in \phi(\pi)\) and \(\pi \in \Pi\) is empty\).

An execution graph with respect to a root configuration and an operation model is a directed labeled graph defined inductively as follows: i) the root configuration is a node in the graph, ii) a configuration \(C'\) is in the graph if \(C \xrightarrow{s} C'\) for some \(C, \pi, s\). A configuration \(C'\) is reachable from a configuration \(C\) if there is a path in the execution graph from \(C\) to \(C'\).

### 5 Analysis of Properties

In this section first we show that persistence and uniqueness are guaranteed by our formalism. Then we describe our results regarding the complexity of checking arrival properties. Henceforth, we refer to our formalism as OpMod.
5.1 Persistence and Uniqueness

In this section, we define uniqueness and persistence and show that our formalism guarantees these properties.

Informally, an artifact instance is unique with respect to an operational model and a root configuration if it is always either in a repository or referenced by a task. Let \( \mathcal{F} = (A, R, T) \) be an operational model in \( \text{OpMod} \), \( C = (\Lambda, \Pi, \phi, \gamma) \) be a root configuration of \( \mathcal{F} \), \( x \in \Lambda \) be an artifact instance of type \((L, P, M)\), \( R \) be a repository in \( \mathcal{R} \). We say \( x \) can arrive into \( \mathcal{R} \) with respect to \( \mathcal{F} \) and \( C \), if there is a reachable configuration \( C' = (\Lambda', \Pi', \phi', \gamma') \) where \( y.id = x.id \) for some \( y \in \Lambda' \). We say \( x \) is always accessible with respect to \( \mathcal{F} \) and \( C \) if there is a reachable configuration \( C' = (\Lambda', \Pi', \phi', \gamma') \) such that there exists an artifact instance \( y \in \Lambda' \) where \( x.id = y.id \) and one of the following is true: (i) \( y \in \gamma(R) \) for some \( R \in \mathcal{R} \); (ii) \( y \in \phi'(\pi) \) and \( y.id = \pi(z) \) for some \( \pi' \in \Pi' \) and \( z \in L \). Finally, \( x \) is persistent with respect to \( \mathcal{F} \) and \( C \), if \( x \) is never destroyed and always accessible with respect to \( \mathcal{F} \) and \( C \).

Informally, an artifact instance is unique with respect to an operational model and a root configuration if it never appears in two places throughout its lifetime. Formally, \( x \) is unique with respect to \( \mathcal{F} \) and \( C \), if there is no reachable configuration \( C' = (\Lambda', \Pi', \phi', \gamma') \) such that there exists an artifact instance \( y \in \Lambda' \) where \( x.id = y.id \), and one of the following is true: (i) \( y \in \gamma(R_1) \) and \( y \in \gamma(R_2) \), for different \( R_1, R_2 \in \mathcal{R} \); (ii) \( y \in \phi'(\pi) \) and \( y \in \phi'(\pi') \) for different \( \pi', \pi'' \in \Pi' \); (iii) \( y \in \gamma(R_1) \) and \( y \in \phi'(\pi) \) for \( R_1 \in \mathcal{R} \), \( \pi' \in \Pi' \).

Our formalism doesn’t contain any deletion operation that can destroy a artifact. Therefore, an artifact never disappears. An artifact instance could become “lost” and inaccessible if in a task it would be possible to overwrite a variable referencing an artifact instance. However, our formalism doesn’t allow this to happen. In addition, our formalism doesn’t contain any operation that can create an exact copy of an artifact instance. An artifact instance could appear in two locations if a task could check in an artifact instance in multiple repositories, or if two tasks could check out the same artifact instance. The former case is prevented because the semantics of checkIn action guarantees that every checkIn is preceded by a new or checkOut or checkAnyOut action. The semantics of check-out actions prevents the later case.

Therefore, we have the following result.

Theorem 5.1 For each operational model \( \mathcal{F} \) defined in \( \text{OpMod} \), for each root configuration \( C \) of \( \mathcal{F} \), and for each artifact instance \( x \) in \( C \), \( x \) is unique and persistent with respect to \( \mathcal{F} \) and \( C \).

5.2 Verification of Arrival Properties

Informally, an artifact can arrive into a repository if there is a way to process the artifact so that it eventually appears in a repository. Let \( \mathcal{F} = (A, R, T) \) be an operational model in \( \text{OpMod} \), \( C = (\Lambda, \Pi, \phi, \gamma) \) be a root configuration of \( \mathcal{F} \), \( x \in \Lambda \) be an artifact instance of type \((L, P, M)\), \( R \) be a repository in \( \mathcal{R} \). We say \( x \) can arrive into \( \mathcal{R} \) with respect to \( \mathcal{F} \) and \( C \), if there is a reachable configuration \( C' = (\Lambda', \Pi', \phi', \gamma') \), and an artifact instance \( y \in \Lambda' \) such that \( x.id = y.id \) and \( y \in \gamma'(R) \).

We first prove that the proposed formalism is computationally very expressive. This leads the verification of the arrival properties to be unsolvable. We obtain this result using a restricted version of the formalism. Let \( \text{OpMod}^{\phi, \gamma} \) represent the operational models defined in \( \text{OpMod} \) where \( \text{dom}_{\leq} \), \( \text{dom}_{=} \) are finite, \( \text{dom}_{\text{id}} \) is infinite, only guard expression allowed is a scalar comparison, and no data function type is used. The next result shows that answering an arrival property is as difficult as answering the halting problem of Turing Machines.

Lemma 5.2 For every Turing Machine \( M \) and an input \( w \), there exists an operational model \( \mathcal{F} \) defined in \( \text{OpMod}^{\phi, \gamma} \) and a root configuration \( C \) of \( \mathcal{F} \) such that \( M \) halts on \( w \) if and only if an artifact instance \( x \) in \( C \) can arrive into a repository \( \mathcal{R} \) with respect to \( \mathcal{F} \) and \( C \).

Proof: [Sketch] The main idea is to show that a Turing machine can be simulated with an operational model. Each machine cell can be represented with an artifact instance. The connection of each cell to its neighbor cells can be represented with 2 attributes of type \( \text{dom}_{\text{id}} \), pointing to left and right cells. Cell contents can be stored in another artifact attribute of type \( \text{dom}_{\leq} \). The current state of the machine can be stored as another artifact instance with one attribute. All the cells are stored in the same repository except the current cell, which is stored in a separate repository. \( M \)'s finite control can be represented with a task. The task checks out the current cell, and the current state, and depending on the content of the cell, it makes the decision about the next cell to be current, and makes the appropriate check-ins and check-outs to be ready for the next configuration. If the finite control specifies that \( M \) should halt, then the task moves an artifact to a special repository. Therefore, an artifact arrives into the repository if and only if \( M \) halts. ✷

Lemma 5.2 leads to the following result:

Theorem 5.3 Given an operational model \( \mathcal{F} \) defined in \( \text{OpMod} \), a root configuration \( C \) of \( \mathcal{F} \), an artifact instance \( x \) in \( C \), and a repository \( \mathcal{R} \) in \( \mathcal{F} \), it is not decidable to check if \( x \) can arrive into \( \mathcal{R} \) with respect to \( \mathcal{F} \) and \( C \).

Next we show that arrival properties are decidable in a restricted version of \( \text{OpMod} \). Let \( \text{OpMod}^{b, T} \) represent the operational models that are defined in \( \text{OpMod} \) such that the number of artifacts that can be created (i.e., the number of invocations of the action \text{new} is bounded) and the only guard expression allowed is \text{true}.

An arrival property with respect to a model in \( \text{OpMod}^{b, T} \), a root configuration can be checked with a
reachability analysis on the execution graph. During the construction of the execution graph, the parameters of the reset actions should be replaced with all possible values from the domains to make sure that the reachability analysis works on all possible cases. Since the domains are infinite, the number of configurations can be infinite. However, it can be shown that the values of attributes and variables do not affect the arrival properties when the only guard allowed is true. Combining this observation with the restriction that there can be only bounded number of artifact instances leads to the conclusion that we only need a bounded number of values when we construct the execution graph to check an arrival property. Therefore, we have the following result:

**Theorem 5.4** Given an operational model $\mathcal{F}$ defined in OpMod$^{b, I}$, a root configuration $C$ of $\mathcal{F}$, an artifact instance $x$ in $C$, and a repository $R$ in $\mathcal{F}$, it is decidable to check if $x$ can arrive into $R$ with respect to $\mathcal{F}$ and $C$ in EXPTIME.

### 6 Conclusion and Future Work

We believe the results we presented in this paper will help the development of automatic static analysis tools for operational models. Businesses today have to quickly respond to changing business requirements. Having a formal model for business operations not only can help in the design of accurate implementations but also can enable the ability to analyze the effects of a change in the business operations. Such an analysis would ease the evolution of business operations and facilitate the quick response to the new business requirements. Another direction this work can lead to is the development of formal analysis techniques for business performance management. Artifacts provide a good granularity in terms of a unit of data that a business processes; this facilitates the formulation of many performance related questions at the right granularity with respect to a business user such as how many guest checks the restaurant can process, how long it would take the restaurant to respond to changing business requirements. Having a formal model for business operations not only can help in the design of accurate implementations but also can enable the ability to analyze the effects of a change in the business operations. Such an analysis would ease the evolution of business operations and facilitate the quick response to the new business requirements. Another direction this work can lead to is the development of formal analysis techniques for business performance management. Artifacts provide a good granularity in terms of a unit of data that a business processes; this facilitates the formulation of many performance related questions at the right granularity with respect to a business user such as how many guest checks the restaurant can process, how long it would take the restaurant to process a kitchen order from creation to archival.

**Acknowledgements:** The authors benefit from the conversations with Kumar Bhaskaran, Henry Chang, Rick Hull, Oscar H. Ibarra, Jun-Jung Jeng, Santhosh Kumaran, Anil Nigam, and Frederick Wu.

**References**


