Technical Report No. NJU-SEG-2011-IC-004

Preservation of Integrity Constraints by Workflow

Xi Liu, Jianwen Su, Jian Yang

Postprint Version. Originally Published in: Lecture Notes in Computer Science 7044, Springer, 2011, pp. 64–81.

Preservation of Integrity Constraints by Workflow

Xi Liu[1,2,3,⋆], Jianwen Su[3,⋆⋆], and Jian Yang[4]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, China
[2] Department of Computer Science and Technology, Nanjing University, China
[3] Department of Computer Science, University of California at Santa Barbara, USA
[4] Department of Computing, Macquarie University, Australia
liux@seg.nju.edu.cn, su@cs.ucsb.edu, jian.yang@mq.edu.au

Abstract. Integrity constraints on data are typically defined when workflow and business process models are developed. Keeping data consistent is vital for workflow execution. Traditionally, enforcing data integrity constraints is left for the underlying database system, while workflow system focuses primarily on performing tasks. This paper presents a new mechanism that turns a workflow into an equivalent one that will preserve integrity constraints. For a given workflow schema (or model) and a given set of data integrity constraints, an algorithm developed in this paper *injects* additional conditions into the workflow schema that restricts possible execution paths. The modified workflow will guarantee data consistency (i.e., satisfaction of the integrity constraints) whenever the workflow updates the database(s). In addition, we show that our injection mechanism is "conservative complete", i.e., the conditions inserted are weakest possible. By making workflow execution self-behaving, enforcing integrity constraints over multi-databases is avoided, and constraints specific to a workflow can also be enforced effectively. Mechanisms such as this enhance independence of workflow executions from the environment—a much desired property.

1 Introduction

Data integrity is the assurance of data correctness, consistency and completeness. From the database perspective, data integrity can be imposed within a database at its design stage through the use of standard rules and procedures, and maintained through the use of error checking and validation routines [2]. Data is the most important asset for any business to make decisions and gain global competitiveness. Decisions made on data that lack integrity can result in losing opportunities and even losing business.

Database management systems (DBMSs) are developed for storing and managing data that is generated and updated by various applications. Workflow systems are an important class of software systems that manage organizational business processes and normally utilize database systems for storing data, executing tasks, and logging. Workflow has been studied for over a decade [16]. Recently with the emerging web service technology, notations and specifications for workflows have been developed such as

⋆ Supported in part by National Natural Science Foundation of China (No.90818022 and No.61021062) and a grant from China Scholarship Council.

⋆⋆ Supported in part by NSF grant IIS-0812578 and a grant from IBM.

BPMN, BPEL, YAWL, etc. These workflow models mostly focus on the aspect of task flow control and completely rely on the underlying database systems to take care of data integrity. However an enterprise workflow system can run across different agencies, departments and organizations, thus it needs to interact with different databases. Take an online shopping workflow as an example, it may need to communicate with a customer database that is only logically integrated from databases of different branches and other partner companies. Distributed DBMS technology does not provide a satisfactory solution in enforcing effectively data integrity defined across multiple database systems. Even when a DBMS detects a violation, it is often difficult to locate the origin in a workflow that causes the error. Also, these underlying databases can be shared by many applications and workflow systems. On the other hand, there are data integrity constraints specific to individual workflow, i.e., they are "local" to the workflow in question. It is not appropriate to enforce such local constraints on databases shared among different applications including other workflows. With the current trend of using "cloud" as the outsourcing facility for data storage and management, pushing local data integrity constraints into a shared database system may result in undesirable effects. Moreover, it is unclear that cloud would realize mechanisms to maintain data consistency in loosely coupled databases [8, 13].

A database system can only check/validate data integrity. It still relies on applications and workflow to produce the correct data and updates, i.e., adhered to integrity constraints. Therefore, in a complex workflow system interacting with distributed databases, it will become an obstacle to always let the database systems check data integrity and come back to the workflow to make necessary corrections for it to proceed.

To overcome the above discussed problems, we propose a mechanism to make a workflow self-behaving in terms of data integrity. The key novelty is to modify a workflow schema by *injecting* certain conditions according to the defined integrity constraints to guard against inconsistent updates. The data integrity is therefore guaranteed within the workflow, and we further gain the independence of workflow execution from the underlying database systems concerning workflow related data updates.

We develop Integrity Preservation Mechanism (IPM) based on a recent artifact-centric workflow model of [17]. The concept of artifact-centricity in workflow modeling was introduced in [24]. There have been increased studies on design and modeling using artifact-centric [5, 6, 17, 22] or other data-aware approaches [12, 21]. The technical development of this paper uses the artifact-centric modeling language GSM (*Guard-Stage-Milestone*) [17]. The language is a declarative meta-model using event-condition-action rules to capture business stakeholders' view [9, 18]. We develop a formal model to specify the execution of GSM workflow based on transition systems and the Z notation [25]. Specified integrity constraints are ensured by strengthening the guard of the operations in the execution workflow schema to prohibit updates that may violate integrity. This process is called *guard injection* in this paper.

To make guard injection work properly, the injection must be *strong enough* to prevent any integrity violations and *weak enough* to allow some or even all correct executions to proceed. The technical challenge is to formulate the appropriate balance in the injection algorithm design.

```
Customer(custid PRIMARY KEY,
         email NOT NULL,
         addr,
         UNIQUE(email) )

Ship(shipid PRIMARY KEY,
     ordid NOT NULL,
     addr NOT NULL,
     name NOT NULL,
     from NOT NULL,
     ship_stat,
     FOREIGN KEY(ordid)
         REFERNECES Order )
```

```
Inventory ( invid PRIMARY KEY,
            prod, avail_qty, loc)

Order(ordid PRIMARY KEY,
      custid NOT NULL,
      invid NOT NULL,
      shipid, qty, ord_stat,
      FOREIGN KEY(custid)
          REFERNECES Customer
      FOREIGN KEY(invid)
          REFERNECES Inventory
      FOREIGN KEY(shipid)
          REFERNECES Ship )
```

Fig. 1. Key artifacts in EzMart

This paper makes the following technical contributions.

1. We formulate a new technical problem of preserving integrity constraints by modifying workflow specifications, develop an algorithm for solving this problem, and prove the correctness of the algorithm.
2. We introduce the concept of "conservative runs" and show that our solution is also "conservative complete", i.e., injections are always weakest possible.
3. In carrying out this work, we also define a formal transition-system semantics for GSM (whose alternative semantics were developed recently [9, 18]).

We note here that although IPM is based on GSM, the methodology and techniques developed in this paper can be easily applied to other workflow specification languages supporting logical data models. In particular, IPM works as long as the action effect can be formulated as a transition system (and the workflow execution is guarded).

The remainder of the paper is organized as follows. Section 2 motivates the problem and illustrates GSM with an example. Section 3 sketches a formal semantics for GSM. Sections 4 and 5 are devoted to the injection algorithm and correctness proof, resp., with the concepts of soundness and conservative completeness included in Section 5. Section 6 reports on related work, and Section 7 concludes the paper. Due to space limitation, we omit detailed formalisms and technical proofs in the paper, and include them in an online appendix [23].

2 A Motivating Example and GSM

In this section, we illustrate the main problem with an example workflow. The example is specified in the declarative artifact-centric workflow model GSM [17], which provides the technical setting for this paper.

2.1 The EzMart Workflow

In an online shopping center "EzMart", a registered customer can buy products and the purchased items are delivered to the customer's address. Modeled with an artifact-centric approach [6], EzMart contains four artifact classes: *Customer*, *Order*, *Ship*, and

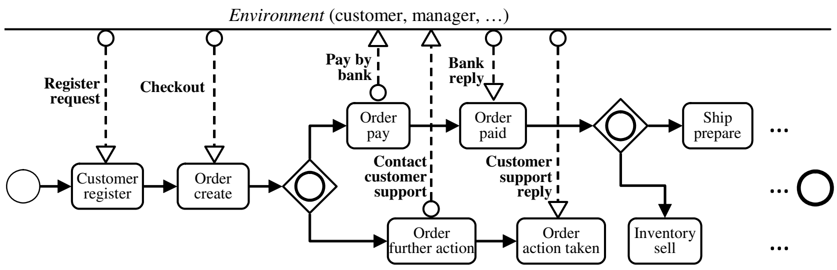

Fig. 2. The EzMart workflow in BPMN-like notation

Inventory. The artifacts are structured as relations as shown in Fig. 1, where *ord_stat* can be one of "CREATE", "INVUPD", "CANCEL", or "RETURN", and *ship_stat* can be one of "PREPAR", "SHIPIN", "FINISH", or "FAILED".

Fig. 2 shows a part of EzMart which is a typical online store process.[1] The customer first registers, can then select products and proceed to checkout. An order is created when the checkout request is made and the customer pays the order using an online bank service. When the order is paid, a shipment process starts and it completes when the package is delivered to the customer. After an order is made, the customer may contact the customer support to take further action(s) on the order, and the order may be returned or canceled (and also possibly changed to other status), the order status is updated accordingly. The back-end inventory management will calculate the available quantity as an order is paid. When the quantity is too low, the inventory manager is notified and a replenishment process starts that will eventually update the quantity (not showed in Fig. 2).

2.2 GSM Specification of EzMart

We now specify key components of EzMart using the workflow language GSM [17]. GSM models complex business process with globalization and out-sourcing in a declarative fashion. The behavior and constraints of business operations are specified in event-condition-action rules.

There are two key constructs in GSM: the information model and the lifecycle model. The former consists of the artifacts and their data attributes (as described in Subsection 2.1). The latter is specified using "stages" consisting of guards, milestones, and stage bodies. Intuitively, a *stage* represents a phase of processing of an artifact. A stage is entered if its *guard* is true, and ends when a *milestone* is accomplished (a condition becomes true). Fig. 3 shows a specification of EzMart in GSM that extends the BPMN workflow shown in Fig. 2. In Fig. 3, a stage (body) is shown as a rectangular with round-corners, a diamond on a stage is the guard (diamond with a "+" in the middle represents the corresponding stage will create a new artifact instance), and a circle on a stage is the milestone (a circle with a bullet indicates a *finish* milestone, a milestone that can complete a lifecycle).

[1] In Fig. 2, we use the inclusive gateway of in BPMN, denoted by a diamond with a cycle in the middle. Such a gateway allows one or more branches following to be taken.

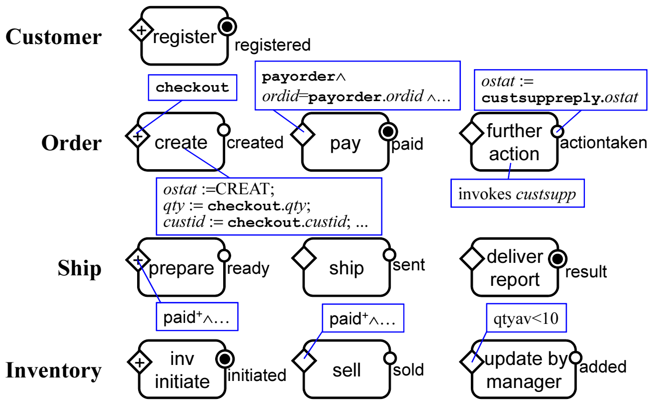

Fig. 3. GSM lifecycle model of EzMart

We focus on artifact class *Order* to illustrate how the GSM model of EzMart works. When the customer is ready to checkout, a `checkout` event is sent to EzMart. An *Order* artifact is then created by stage create, where *ord_stat* is set to "CREATE", *custid* and *ordid* (reference to associated customer and inventory, resp.) and the *qty* are set according to the content of the *triggering* event `checkout`.

To pay the order, the customer sends a `payorder` event with *ordid* matching the artifact ID. The stage pay then opens, and bank service is invoked to pay the order. When the bank replies, the finish milestone paid is achieved.

After the order is made, the customer can request some further actions on the order. If the customer does so, stage further_action opens and customer support human task *custsupp* is invoked. The milestone actiontaken is achieved by the reply event of customer support. This reply takes the immediate effect on milestone actiontaken to change the status of the order, e.g. mark the status of the order as canceled.

When the order is paid, the *control event* paid^+ (achieving the milestone paid) can trigger stages prepare of *Ship* and sell of *Inventory*. Furthermore, in stage sell, the *ord_stat* of the *Order* artifact identified by the ID retrieved from the event paid^+ is assigned to "INVUPD" (i.e. $Order(\text{paid}^+.ordid).ostat := \text{INVUPD}$).

Stage guard conditions and milestones are formulas on events and attributes (stage and milestone status is not used). In the remainder of the paper, the term "sentry" is used to refer to conditions for both guard and milestones.

2.3 Integrity Constraints

Artifacts are stored in a database (conceptually). Data in workflow are a representation of the reality and thus integrity constraints arise naturally [6]. Some integrity constraints in EzMart (e.g. not-null, keys and foreign keys) are already shown in the artifact relation definitions in Fig. 1. There are additional integrity constraints. The constraints on ***attribute content*** restrict the domains of attribute values. For each *Order* artifact, the quantity is greater than 0, i.e. $qty > 0$; for *Ship* artifacts, sending address must be different from the delivery address, i.e. $from \neq addr$; for *Inventory* artifacts, the available quantity is non-negative, $avail\_qty \geq 0$.

In addition to the constraints on attributes of a single artifact, there are further *business specific constraints*:

- ***Status constraint***: Given an artifact s of *Ship*, if the shipment has started but not yet finished, the associated order cannot be canceled nor returned. That is, if $s.ship\_stat$ is neither "FINISH" nor "FAILED", and there is an artifact o of *Order*, such that $s.ordid = o.ordid$ and $o.shipid = s.shipid$, then the order status $o.ord\_stat$ must not be "RETURN" nor "CANCEL".
- ***Address-name constraint***: Given an artifact of *Order*, the delivery address *addr* and recipient's name *name* of the associated *Ship* artifact must match the address *addr* and *name* of the associated *Customer* artifact.
- ***Ship-from constraint***: Given an artifact of *Order*, the sending address of the shipment must match the inventory warehouse location *loc*.
- ***Ship-order reference circle***: Given an artifact of *Order*, the *Ship* artifact referenced by attribute $o.shipid$ must also reference back to o, and *vice versa*.

2.4 Enforcing Integrity Constraints: A Challenge

Traditionally workflow systems rely on the underlying database system to ensure data consistency. However, in reality the data are quite likely stored and managed distributedly. Artifacts in a single artifact class may be stored in several databases. Assume that EzMart combines two old shopping centers that maintain their own customer databases. Some integrity constraints in EzMart, e.g. the candidate key on *Customer*, cannot be handled properly and easily by one database system alone [14].

The recent trend of cloud computing and SOA brings other opportunities: (1) data management of EzMart can be outsourced and the service provider may "pack" similar data from EzMart and other applications together, (2) the customer data EzMart uses may be owned by a separate data service provider who may not respect data integrity constraints from EzMart. Consider the repository for *Order* of EzMart that shares the same actual data with electronic order database of other companies. The data service provider has to keep the data of EzMart from the other companies as well as maintaining constraints from different applications. This results in high complexity and expenses. In general, elevating integrity constraints local to one workflow to global for the data service provider is problematic. For example, other applications may not require quantity in the order to be strictly positive (cf. attribute content constraint of EzMart).

It is desirable for a workflow to block its own updates if they violate integrity constraints. Consider the attribute content constraint on *Order* that requires for each *Order* artifact o, $o.qty > 0$. In Fig. 4, the stage create uses the triggering event `checkout` to assign *qty*. Then if we strengthen the sentry of the guard to allow only the event with `checkout`$.qty > 0$ to pass, the constraint on *Order* cannot be violated by the update from the stage. To generalize this idea, associated data integrity constraints should be preserved *within* EzMart by strengthening the guard condition— the *guard injection*.

As an extreme, the simplest and effective injection is to inject FALSE to the guard of every stage. Then no execution would violate the constraints—because there will be

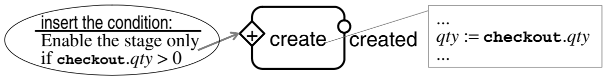

Fig. 4. Prevent the violation by strengthen the sentry condition

no execution at all. To make the injection useful, we need to make sure the injected constraints must be weaker or even the "weakest". The injection should block all executions that can violate the constraints but should also allow as many executions as possible that preserve data consistency.

This paper develops a technical approach that calculates injection according to the constraint set and actions in stage bodies. The approach is proved to be both "sound" (strong enough to prevent violations) and "conservative complete" (weak enough to have a useful workflow).

3 A Formal Semantics of GSM

In order to analyze GSM workflow for possible injection, it is necessary to formalize its semantics. In this section, we define the execution model (an operational semantics) of GSM specifications. First we give an intuitive explanation of GSM execution. Then we present a transition system semantics for GSM. The formalism is inspired (but not restricted) by the Z notation [25]. Our transition system semantics is complementary to recent GSM operational semantics presented in [9, 18]. Our semantics focuses on manipulation of data and system variables, and forbids the concurrency between atomic stages. A brief comparison can be found in the online appendix [23].

3.1 Intuitive Explanation

The GSM execution model was initially described in [17], and further developed in [18]. A workflow starts with no artifacts. An artifact is created when a create-instance stage opens. A stage opens if its sentry is satisfied and then actions defined in the stage body starts to execute. For example, stage update_by_manager in EzMart opens when attribute *avail_qty* of the *Inventory* artifact is less than 10. If the stage is to create instance, a new artifact is created, e.g. create stage creates a new *Order* artifact.

A milestone is achieved when its achieving sentry is satisfied and its belonged stage is open. For example, actiontaken is achieved when stage further_action is opened and the reply event from task *custsupp* comes (i.e. the head event in the external event queue is reply from *custsupp*). A milestone is invalidated when its invalidating sentry is satisfied, i.e. the milestone changes to or stays in the status of not achieved.

A stage can also reopen, when its sentry is satisfied again. For example, stage further_action can run several times as long as the event `takeaction` is received. Our workflow model extends the GSM slightly with the notion of "finish" milestones, such as registered, paid, etc. And only when all of the finish milestones are achieved can we say a workflow execution finishes (formal and strict definition in Section 5).

When the status of a stage or a milestone changes, a control event is generated. The control event can also be used as a triggering event. An example is the triggering event of sell. The stage needs a control event paid^+ to open, and paid^+ denotes the paid milestone's achieving.

3.2 GSM Transition Systems

In our execution model, a GSM workflow is a transition system consisting of a state space (a set of states), an initial state and a set of operations (or transitions). The state space is a set of all possible "snapshots" of artifacts, each is called a *state* which specifies, at a specific time during the execution, the attribute value, the status of stage and milestones and event queues. The *initial state* is a special state that the transition system starts on, where no artifacts exists in the transition system, and event queues are empty.

Operations are transitions from one state to the next, identified by "operation signatures", and the transition enabling condition (called the "guard") and state changes made by the transition (called the "actions") are specified. There are the following six types of operations:

– *Open* : opens a stage, if the stage sentry is satisfied and no other stage is open. When the stage opens, all of its milestones are set to be not achieved. New instance is created if the stage is a create-instance stage.
– *Body* : executes the stage body.
– *AchieveClose* : achieves a milestone. The operation is enabled when the achieving sentry of the milestones is satisfied, and the stage containing the milestone is opened. When the milestone is achieved, its stage is closed. And if the milestone is triggered by some external events, immediate effect is taken on the attributes.
– *Invalid* : invalidates a milestone (changes the milestone status to not achieved).
– *DeCQ* : removes the head event from control event queue. This operation has a lower priority than the above four types of operations.
– *DeEQ* : removes the head event from external even queue. This operation has the lowest priority. That is, only when none of the above five types of operations can be enabled, can *DeEQ* be enabled.

Status change of stage opening and closing, milestone achieving and invalidating generates control events, which are also control events to be added to the event queue (used to trigger other stages or milestones in the workflow). Parameters of the signature for each *Open* operation are the stage name and (if the stage is a create-instance stage) the artifact ID, for *Body* are the stage name and artifact ID, for *AchieveClose* and *Invalid* are the milestone name and artifact ID, and no parameter for *DeCQ* and *DeEQ*.

Due to space limitation, only the state space and *Open* operation (for a create-instance stage) are presented, an example is given to briefly explain the operations *AchieveClose*. A complete formalism of state space and operations is included in [23].

Given a GSM workflow *AP*, its transition system is denoted by TS_{AP}. The state space is specified by the construct *STATE* in Z notation, shown in the left column of Fig. 5.

In the state space each artifact class α_i is represented as a table consisting the artifact ID, the data attribute value (denoted by $x, y, \ldots$), and stage and milestone status (denoted by stage or milestone names, such as s,m,...). And the set of all artifact classes in *AP* is $\mathbf{A} = \{\alpha_i \mid i \in 1 \, . \, . \, n\}$. For any artifact class α, we use $S(\alpha)$ and $M(\alpha)$ to denote the sets of all stages and milestones of α, resp.

The state variable *XOp* is a finite set of signatures of possible next operations, where *OPSIG* is the type of operation signatures. Variable *eq* and *cq* are two queues of external events and control events, resp., where *ExtEv* and *IntEv* are respectively the type of external events and control (internal) events. State variables that do not immediately help in understanding the execution model are omitted.

STATE

$\alpha_1(id, x, y, \ldots) : \mathit{ArtifactClass}$
$\alpha_2(\ldots, , \mathsf{s}, \mathsf{m}, \ldots) : \mathit{ArtifactClass}$
$\vdots$
$\alpha_n(\ldots) : \mathit{ArtifactClass}$
$\mathit{XOp} : \mathbb{F}\,\mathit{OPSIG}$
$\mathit{eq} : \mathrm{seq}\,\mathit{ExtEv}$
$\mathit{cq} : \mathrm{seq}\,\mathit{IntEv}$
$\vdots$

Open(create)

$\mathit{new} = \mathit{Order}(\mathit{newid}(), \mathsf{null}, \ldots,$
$\quad \text{TRUE}\ /^*\mathsf{create}^*/, \text{FALSE}, \text{FALSE}, \ldots)$

$\mathit{Open}(\mathsf{create}) \in \mathit{XOp}$
$\forall \beta : \mathbf{A};\ id : \mathit{ID};\ \mathsf{t} : S(\beta) \bullet \beta(id).\mathsf{t} = \text{FALSE}$
$(\mathit{head}\ \mathit{eq})$ `isevent checkout`
$\ldots$

$\mathit{Order} := \mathit{Order} \cup \{\mathit{new}\}$
$\mathit{XOp} := (\mathit{XOp} - \{\mathit{Open}(\mathsf{create})\}) \cup$
$\quad \{\mathit{Body}(\mathsf{create}, \mathit{new}.id)\}$

Fig. 5. State space and an example operation

We give an example state s of TS_{EzMart}:

- $s.\mathbf{A}$ is the set of tables of artifacts of classes *Customer*, *Order*, *Ship* and *Inventory*.
- $s.\mathit{XOp} = \{\mathit{Open}(\mathsf{create})\}$, the set of the open stage operation of create of *Order*.
- $s.\mathit{eq} = \langle$`checkout`$(\ldots), \ldots\rangle$ and $s.\mathit{cq} = \langle\,\rangle$ (empty queue).

On the initial state, all artifact class tables are empty, next operation set (*XOp*) is the set of *Open* operations for create-instance stages, external and control event queues (*eq* and *cq*) are both empty. As for EzMart, *init*.*XOp* is the set of *Open*(register), *Open*(create), *Open*(prepare) and *Open*(inv_initiate).

Operations are defined using the schema extended from Z notation [25].[2] Operation *Open* is responsible to handle stage opening. We use *Open*(create) of *Order*, specified in the right column of Fig. 5, as an example of the operation to open a create-instance stage. Let s be the current state as the example state given above, and s' be the next state after the transition specified by the operation *Open*(create). Suppose $s.\mathit{Order}$ has one row: $\mathit{Order}(\mathrm{ord001}, \mathrm{cust002}, \ldots)$. If there is no stage being open on s, then s satisfies the guard of *Open*(create). A new *Order* artifact, denoted by the local variable *new*, is created, where the ID is assigned using a system new ID generator, all other attributes are assigned to null and statuses of all stages and milestones are set to FALSE except the status of create is set to TRUE. Let's assume the new ID generator gives ord002 on s. Then, after this operation, $s'.\mathit{Order}$ has two rows,

$$\mathit{Order}(\mathrm{ord001}, \mathrm{cust002}, \ldots);\ \ \mathit{Order}(\mathrm{ord002}, \mathsf{null}, \ldots, \text{TRUE}, \text{FALSE}, \text{FALSE}, \ldots)$$

and $s'.\mathit{XOp} = \{\mathit{Body}(\mathsf{create}, \mathrm{ord002})\}$. Since the open stage event of create is not used in EzMart, control event queue cq is kept unchanged.

[2] Readers who are familiar with Z can find out our notation still follows the fundamental idea of Z, and the extension is only "syntactical" to make our specification easier.

We group achieving of a milestone and closing of its stage in *AchieveClose*. Consider the operation of achieving milestone actiontaken. The signature of the operation is $AchieveClose(\text{actiontaken}, ordid)$. Suppose that on the current state s, $s.XOp$ contains $AchieveClose(\text{actiontaken}, \text{ord002})$, $s.Order(002).\text{pay} = \text{TRUE}$, $s.Order(002).\text{paid} = \text{FALSE}$, and $s.eq = \langle \texttt{custsuppreply}(\dots) \rangle$, where event `custsuppreply` is the reply from customer support task *custsupp*. Then this operation can be enabled with *ordid* taking the value of ord002. As a result, on the next state, artifact $s'.Order(\text{ord002})$ is updated as $\text{pay} = \text{FALSE}$, $\text{paid} = \text{TRUE}$, and *ostat* is set according to `custsuppreply`.*ostat* — the immediate effect of the event.

4 Guard Injection

Based on the GSM execution model, we explore in this section our approach to enforce integrity constraints. The key idea is to inject conditions according to the specified constraints and the stage to "block" possible violations. We first define constraints and some needed notions, and then present the algorithm for guard injection.

Our problem is similar to but different from checking integrity constraints in distributed databases where the exact changes to the database are known [14, 19]. Our problem considers workflow specifications when the updates to the database are unknown but parameterized. A key idea of [14, 19] is to "look forward" at the "post-condition" of the update and check locally if the constraint with respect to the post-condition can be satisfied without looking at database(s). (If not, databases are consulted to check the integrity constraints.) The injection technique developed in this paper "looks backward" to calculate "weakest" precondition of stage and ensures that *potentially* executed updates would never violate the constraints.

4.1 Integrity Constraints

In this paper, each integrity constraint κ is defined in the following form (cf [2]):

$$\kappa = \forall \mathbf{x} \,.\, (\, \phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \,.\, \psi(\mathbf{x}, \mathbf{y})\,) \tag{1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are finite vectors of variables with no repetition, $\mathbf{x}$ is nonempty (while $\mathbf{y}$ can be empty), and Formula ϕ and ψ are nonempty conjunction of artifact relation atoms and comparison atoms of the form $x \circ y$, where x is a variable and y is a variable or constant and $\circ$ denotes operators: $=, \neq, \geq, \leq, >$ and $<$. Variables in $\mathbf{x}$ and $\mathbf{y}$ are artifact IDs or data attributes of artifacts in *AP*. There is *at least one* artifact relation atoms in ϕ. Formula ϕ uses all variables in $\mathbf{x}$ and ψ uses all variables in $\mathbf{y}$.

The conjunctive atoms in ϕ are *premises* and the right hand side of the arrow is referred to as the *consequent*. For simplicity, we assume all constraints are free from trivial atoms (equivalent to TRUE or FALSE). Variables in $\mathbf{x}$ are referred to as $\forall$-quantified variables, while the ones in $\mathbf{y}$ as $\exists$-quantified variables.

For example, the attribute content and not-null constraint on *Order* and status constraint are given in Section 2 and we repeat here:

Attribute content and not-null constraint on** Order**, κ_{attr}: For each *Order* artifact, neither of the references *custid* and *invid* is null, and the quantity is larger than 0, i.e. $qty > 0$;

Status constraint, κ_{stat}: Given an artifact s of *Ship*, if $s.ship\_stat$ is neither FINISH nor FAILED, and there is an artifact o of *Order*, s.t. $s.ordid = o.ordid$ and $o.shipid = s.shipid$, then the order status $o.ord\_stat$ must not be RETURN or CANCEL.

Written in the form of Equation (1), these two constraint formulas are:

$$\begin{aligned}
\kappa_{attr} = \;& \forall\, ordid, custid, invid, shipid, qty, ord\_stat.\\
& \quad Order(ordid, custid, invid, shipid, qty, ord\_stat) \rightarrow\\
& \qquad\qquad custid \neq \mathsf{null} \wedge invid \neq \mathsf{null} \wedge qty > 0\\
\kappa_{stat} = \;& \forall\, ordid, custid, invid, shipid, qty, ord\_stat, addr, name, from, ship\_stat.\\
& \quad Order(ordid, custid, invid, shipid, qty, ord\_stat) \wedge\\
& \quad Ship(shipid, ordid, addr, name, from, ship\_stat) \wedge\\
& \quad ship\_stat \neq \text{FINISH} \wedge ship\_stat \neq \text{FAILED} \rightarrow\\
& \qquad\qquad ord\_stat \neq \text{RETURN} \wedge ord\_stat \neq \text{CANCEL}
\end{aligned}$$

The complete list of formulas for all constraints in EzMart in the form of Equation (1) can be found in [23].

We say a *concerning attribute* of constraint κ is some attribute $\alpha.x$ of some artifact class α, where $\alpha.aid$ is in $\forall$-quantified variables of κ, α is an artifact relation atom in κ, and any one of the following holds:

- there is a constant appearing at the column of x in artifact relation atom of α in κ;
- there is a variable x' appearing at the column of $\alpha.x$ in artifact relation atom of α, and $x' \circ y$ also appears in κ, where y is a constant or any other variables; or
- there is some variable x' appearing more than once in κ and one of its appearances is at the column of $\alpha.x$ in artifact relation atom of α.

The set of concerning attributes of κ is denoted by $CA(\kappa)$.

The sets of concerning attributes of κ_{attr} and κ_{stat} are

$$\begin{aligned}
CA(\kappa_{attr}) &= \{Order.custid, Order.invid, Order.qyt\}\\
CA(\kappa_{stat}) &= \{Ship.ordid,\ Ship.ship\_stat,\ Order.shipid,\ Order.ord\_stat\}\,.
\end{aligned}$$

Given a constraint κ and an attribute $x \in CA(\kappa)$, the set of *writing stages* of x is denoted by $WS(x)$. A writing stage of x is such a stage s that $x \in WriteSet(\mathsf{s})$, where $WriteSet(\mathsf{s})$ denotes the set of attributes that can be written by the body of stage s. The set of *writing milestones* of x is denoted by $WM(x)$. A writing milestone of x is such a milestone m that uses the reply event to update attribute x.

In EzMart, the writing stage of concerning attributes of κ_{attr} is create; and the writing stages and the writing milestone of concerning attributes of κ_{stat} are

Ship.ordid	: prepare	*Ship.ship_stat*	: prepare, ship
Order.shipid	: ship	*Order.ord_stat*	: create, sell, actiontaken

4.2 Calculating Injected Conditions

The algorithm to calculate the injection is now presented. The intuition of the injection is first described. Then the algorithm is given along with examples using attribute content and not-null constraint on *Order* (κ_{attr}) and status constraints (κ_{stat}).

The intuition of the algorithm is to "inject" properly converted constraints into guards of *Open* operations of writing stages and stages of writing milestones of concerning attributes of the constraints. As a result, the guard is strengthened to block all updates that may violate the integrity constraints, but to allow updates that preserve the data integrity.

For writing stages, we analyze the stage body to understand how the updates are made. For example, stage create uses the triggering external event `checkout` to set *custid*, *invid* and *qty* — the concerning attribute of κ_{attr}. The injection is a substitution for the concerning attribute according to the stage update. The injection is replacing concerning variable in κ_{attr} by corresponding content of event `checkout`. Assume there is no violation before the update made by create, if the current state satisfies the injection, the execution of create under such a `checkout` event preserves the data consistency regarding to κ_{attr}. Moreover, such an injection is also weak enough only to block the updates that result in violation. As for the writing milestones, because there is no information about the task reply in the workflow, the injection to its stage's *Open* operation has to be made to the strongest—FALSE.

In the following we present the detail of the algorithm. To begin with, implicit foreign key references are made explicit in the constraint set. A stage, e.g. ship, of one artifact class α, e.g. *Ship*, can write attributes of artifact class β, e.g. *Order.shipid*. Then the foreign key constraint is added to the integrity constraint set, if it is not already specified:

$$\forall\, aid, bid, \mathbf{x}\,.\, \alpha(aid, bid, \mathbf{x}) \wedge bid \neq \mathsf{null} \rightarrow \exists\, \mathbf{y}\,.\, \beta(bid, \mathbf{y})$$

where **x** and **y** are disjoint vectors of other "unrelated" attributes in α and β, resp. In EzMart, all of the foreign key dependencies are already specified.

Algorithm 1. Guard injection

```
Input: $TS_{AP}$ and $\mathbf{K}$
Output: $Inj : \mathbf{S} \rightarrow FO$
Set $Inj(\mathsf{s}) :=$ TRUE for each stage $\mathsf{s}$ in $AP$;
foreach $\kappa \in \mathbf{K}$ and $x \in CA(\kappa)$ do
    foreach $\mathsf{s} \in WS(x)$ do
1       $Inj(\mathsf{s}) := Inj(\mathsf{s}) \wedge$ SUB$(\kappa, \mathsf{s}\,[, aid])$, where
          $aid$ is the artifact ID in $Open(\mathsf{s}\,[, aid])$ in $TS_{AP}$;
    endfch
    foreach $\mathsf{m} \in WM(x)$ do
        $Inj(\mathsf{s}) :=$ FALSE
    endfch
endfch
```

The procedure of injection is given in Algorithm 1. It takes the transition system TS_{AP} of GSM specification *AP* and the set of integrity constraint **K** as input. The output is the injection function *Inj* which maps each stage in *AP* to a first order formula. The idea of the algorithm is simple. For each constraint κ in **K** and for each concerning variable x of κ, if x is going to be written by a stage s, the algorithm injects the converted κ by replacing x with the assignment in the body of s, where the substitution is accomplished by function SUB (given below); if x is going to be written by a reply event triggering a milestone, then FALSE is injected.

The function SUB used in Line 1 of Algorithm 1 is to convert the constraint formula κ according to the stage s and (if s is not a create-instance stage) the artifact ID *aid*. Concerning attributes and artifact relations are replaced according to the assignments in the body of s.

First, in Line 2 of SUB, by function $\mathit{explicitref}(\kappa)$, reference dependency premises are added when the stage writes attributes of another artifact class. The substitution procedure then starts by taking care of each $\forall$-quantified α IDs. The variable x is replaced by the assignment in the stage body (Line 3), and the ID is replaced (Line 4), by $\mathit{newid}()$ if the stage creates new instance; otherwise, by the artifact ID of the updated artifact in the stage assignment (denoted by *lhsid*, which is ID field of head event if the artifact is identified by the event or *aid* otherwise). After the variables are properly replaced, they are removed from the $\forall$-quantified variable list. Note that when the stage is creating an instance, replacing the *id* with $\mathit{newid}()$ also replace the relation atom to TRUE because the new instance is going to be inserted and therefore the relation atom holds. Note that after the substitution, $\beta(\ldots,\mathit{newid}(),\ldots)$ is further replaced by FALSE for any artifact relation β, and $\mathit{id} \circ \mathit{newid}()$ is further replaced by FALSE for any *id* unless $\circ$ is $\neq$. More technical details of SUB can be found in the online appendix [23].

Function. SUB (κ, s[, *aid*]) returns the constraint formula after substitution

```
  $ID^\kappa$ := all $\forall$-quantified artifact IDs of the artifact whose attribute is updated in s;
2 $con := \mathit{explicitref}(\kappa)$;
  $ret :=$ TRUE;
  foreach $id$ in $ID^\kappa$ do
3     $con := con[\mathbf{exp}/\beta(id).\mathbf{x}]$, where $\mathbf{x}$ and $\mathbf{exp}$ are vectors of the same length
        and for any $0 \le i < \#\mathbf{x}$, s.$Body$ contains $\beta(id).\mathbf{x}_i := \mathbf{exp}_i$ ;
4     if s.$creatInst$ = TRUE then
          $con := con[\text{TRUE}/\alpha(id,\ldots)]$, where s $\in S(\alpha)$;
          $con := con[\mathit{newid}()/id]$
      else
          $con := con[\mathit{lhsid}/id]$;
      endif
      $ret := ret \wedge con$
  endfch
  return ret
```

In EzMart, for the attribute content and not-null constraint on *Order* (κ_{attr} in Subsection 4.1), all of the three concerning variables are replaced by the corresponding content of event checkout in stage create. And since the stage creates a new instance, the *Order* artifact relation atom is replaced by TRUE. This gives the substitution result, after trivial reduction,

$$\text{SUB}(\kappa_{attr}, \mathsf{create}) = \text{TRUE} \rightarrow (\mathit{head}\ \mathit{eq}).\mathit{custid} \neq \mathsf{null} \wedge (\mathit{head}\ \mathit{eq}).\mathit{invid} \neq \mathsf{null} \wedge (\mathit{head}\ \mathit{eq}).\mathit{qty} > 0$$

Then for the status constraint (κ_{stat} in Subsection 4.1), the stage sell of *Inventory* sets $Order((head\,cq).ordid).ord\_stat$ to "INVUPD", where by the stage sentry, $head\,cq$ is a paid^{+} event. Therefore we have,

$$\begin{aligned}\mathtt{SUB}(\kappa_{stat}, \mathsf{sell}, invid) = {} & \forall\, custid, shipid, qty, ord\_stat, addr, name, from, ship\_stat\,\boldsymbol{.}\\ & Order((head\,cq).ordid, custid, invid, shipid, qty, \textsc{invupd}) \wedge\\ & Ship(shipid, (head\,cq).ordid, addr, name, from, ship\_stat) \wedge\\ & ship\_stat \neq \textsc{finish} \wedge ship\_stat \neq \textsc{failed} \rightarrow\\ & \quad \textsc{invupd} \neq \textsc{return} \wedge \textsc{invupd} \neq \textsc{cancel}\end{aligned}$$

where *ordid* is replaced by $(head\,cq).ordid$, and *ord_stat* is replaced by the constant INVUPD. And obviously, this is equivalent to TRUE.

After Algorithm 1 completes, injection to stages of *Order* is given as follows. Note that trivial expressions are directly removed and the injected formula is reduced.

$$\begin{aligned}Inj(\mathsf{create}) = {} & \mathtt{SUB}(\kappa_{attr}, \mathsf{create}) \wedge\\ & \exists\, email, addr, name, info, prod, avail\_qty, loc, price\,\boldsymbol{.}\\ & \quad Customer((head\,eq).custid, email, addr, name, info) \wedge\\ & \quad Inventory((head\,eq).invid, prod, avail\_qty, loc, price)\\ Inj(\mathsf{further\_action}) = {} & \textsc{false}\end{aligned}$$

The second conjunct injection on stage create is the one for foreign key constraint (to *Customer* and *Inventory*); injections for ship-order reference circle, address-name, ship-from and status constraint are all reduced to TRUE. Injection on further_action is FALSE because actiontaken milestone of this stage uses the reply event to update *ord_stat*, a concerning attribute of the status constraint κ_{stat}.

The complete list of constraints and injections on EzMart can be found in [23].

5 Soundness and Conservative Completeness

To state the correctness of the injection algorithm, we first define some technical notions. Examples from EzMart are given along the definitions and the technical results of "soundness" and "conservative completeness".

Definition 1 (run and complete run). Let *AP* be a GSM specification, TS_{AP} its transition system. A *run* of TS_{AP} is an alternating sequence of states and operations

$$\rho = s_0 t_0 s_1 t_1 \dots t_{n-1} s_n$$

where $s_0, s_1, \dots$ are states (specified by *STATE*), s_0 is the initial state, and $t_0, t_1, \dots$ are operations (specified by *OPER*), such that for each t_i $(i \geq 0)$, $s_i \models guard(t_i)$.

Let $\mathbf{A}$ be the set of artifact classes in *AP*. A run ρ is said to be *finished* iff state s_n satisfies that for each artifact of $\alpha : \mathbf{A}$ with $id : ID$,

- there is a finish milestone achieved, i.e. $\exists\, \mathsf{m} : M(\alpha)\,\boldsymbol{.}\, \mathsf{m}$ is a finish milestone $\wedge$ $\alpha(id).\mathsf{m} = \textsc{true}$; and
- there is *no* stage being open, i.e. $\forall\, \mathsf{s} : S(\alpha)\,\boldsymbol{.}\, \alpha(id).\mathsf{s} = \textsc{false}$.

In EzMart, each run starts with the initial state and followed by an open stage operation of register, create, prepare or inv_initiate. On state s_1, only the stage body operation of the just opened stage in t_0 can be enabled. State s_2 is the result of the stage body operation. The run is finished if on the last state in the run, milestones registered of all artifacts of *Customer*, paid of all artifacts of *Order*, result of all artifacts of *Ship* and the initiated of all artifacts of *Inventory*, are all achieved, and there is no stage being open.

Given a run, if it does not violates any constraint, we say this run is sound.

Definition 2 (Sound run). Let *AP* be a GSM specification, TS_{AP} be its transition system and **K** a set of integrity constraints on artifacts of *AP*. A run ρ of TS_{AP} is said to be **K**-*sound* iff for each $\kappa \in \mathbf{K}$, κ holds in every state in ρ. When **K** is clear from the context, we simply say ρ is *sound*.

Consider run $\rho_1 = s_0t_0s_1t_1s_2t_2s_3$ in EzMart where t_0 is *Open*(register) and creates a new *Customer* artifact with ID cust001, t_1 is *Body*(register, cust001) which sets *Customer*(cust001).*email* to `abcdef.com`, and t_2 is *AchieveClose*(registered, cust001). We can see that ρ_1 is a finished run, and also sound. Because *email* is not empty, attribute content constraint on *Customer* is satisfied on all of the states in ρ_1. There is only one artifact in the system, the candidate key and foreign keys are also satisfied, and all business specific constraints are also satisfied.

Now consider another run $\rho_2 = s_0t_0 \cdots t_is_{i+1} \cdots$. Suppose on s_i, there is an artifact *Ship*(ship005).*ship_stat* = SHIPIN and t_i is *AchieveClose*(action_taken, ord002). If *Order*(ord002).*shipid* = ship005 and the immediate effect of reply event is to set the *Order*(ord002).*ord_stat* to CANCEL, then status constraint (κ_{stat}, see Subsection 2.1) is violated, because the order ord002 is canceled when the purchased item is still shipping. Therefore ρ_2 is not a sound run.

In business processes, many tasks are third-party services or human tasks. It is not reasonable to assume all tasks will strictly follow some contract. We have to be prepared that external tasks may give unpredictable reply in the domain. To ensure the constraints are never violated, we need to be cautious or conservative. Therefore, in the Algorithm 1, if the reply event may update a concerning attribute of a constraint, FALSE is injected to the associating stage.

Definition 3 (Conservative run). Let *AP*, TS_{AP} and **K** be the same as in Definition 2. A finished run ρ of TS_{AP} is said to be **K**-*conservative* iff ρ is sound and for any constraint $\kappa \in \mathbf{K}$, there is no reply event being used to update any attribute in $CA(\kappa)$. When **K** is clear from the context, we simply say ρ is *conservative*.

Consider a run $\rho_3 = s_0t_0 \cdots s_kt_ks_{k+1} \cdots$, and t_k is *Body*(further_action, ord002). The milestone actiontaken belongs to stage further_action, and is a writing milestone of concerning attributes *ord_stat* of status constraint. Therefore ρ_3 is not conservative.

The transition system of *AP* with injection according to integrity constraint set **K** is denoted by $InjTS_{AP}(\mathbf{K})$. When **K** is clear from the context, $InjTS_{AP}(\mathbf{K})$ is simply written as $InjTS_{AP}$. It is constructed by concatenating in conjunction *Inj*(s) with the original guard of *Open* operation of each stage s. (If *Inj*(s) is equivalent to TRUE, then the operation guard after injection is equivalent to the one before.) The definition of "correct" injection, is given by the notion of "soundness" and "conservative completeness" of transition system with injection, where soundness captures no violation—the injection is strong enough, while conservative completeness allows the maximal behavior under conservative strategy— the injection is weak enough.

Definition 4 (Sound and conservative complete injection). Let *AP*, TS_{AP} and $\mathbf{K}$ be the same as in Definition 2, and *Inj* be the guard injection function. We say the injection *Inj* is *sound* iff each finished run of $InjTS_{AP}$ is sound, *conservative complete* iff each conservative run of TS_{AP} is also a conservative run of $InjTS_{AP}$.

The main property of our algorithm is now stated, a proof can be found in the online appendix [23].

Theorem 1. Given a GSM specification *AP* and a set of integrity constraints $\mathbf{K}$, the transition system with injection, $InjTS_{AP}$, is both sound and conservative complete.

Again, we take advantage of EzMart to illustrate the idea of injection correctness. First, for any run of $InjTS_{\mathrm{EzMart}}$, there is no violation. Take stage create as an example. If the head event of *eq* satisfies the sentry and injection of create, then, because *Inj*(create) uses the assignment in create, after the update made in the body of create, the constraint is still satisfied. And because $Inj(\text{further\_action}) = \text{FALSE}$, this injection can block any possible updates made by the incoming reply event, and therefore ensures the integrity constraints.

Then, suppose there is a conservative run ρ of TS_{EzMart} that is not in $InjTS_{\mathrm{EzMart}}$. If this is because the *Open* operation of create cannot be enabled in the injected workflow, say $(head\ eq).qty = 0$, then *Inj*(create) fails; in ρ after the update made by create, the *qty* of newly created artifact of *Order* is 0 which violates the attribute content constraint on *Order*, and thus ρ cannot be sound. If it is because of the blocking in the injected system of *Open* operation of further_action, then ρ is not conservative. Therefore, the injected workflow $InjTS_{\mathrm{EzMart}}$ is both sound and conservative complete.

6 Related Work

Triggers are a powerful tool to "fix" constraint violations as a reactive means, e.g. [7]. In distributed databases, checking constraints involving remote databases is expensive. It was discussed in [15] to maintain distributed integrity constraint efficiently by reducing the necessity to look at remote databases. It was investigated in [14] to use local data to test conjunctive query constraints with arithmetic comparison. In [19], a similar problem is discussed on conjunctive query constraints with negations. The approach used to generate complete local tests is basically to check constraint containment and calculate local tests with respect to "post-condition" of specific updates. To the contrary, our injection is to calculate the weakest precondition conservatively of potential updates and to ensure that updates never result in violation.

It is also studied that by enhancing the underlying systems, data consistency can be maintained in loosely coupled databases. A framework was developed that uses several communication protocols between different sites to maintain data consistency [13]. When strict consistency cannot be ensured, enforcing the weakened integrity constraints is possible by a rule-based configurable toolkit presented in [8]. While these work construct strong data management systems, our work makes a minimum requirement on underlying DBMSs.

Using preconditions can also be found in [4] and [11]. The idea of finding weakest precondition rooted in [10]. The difference of using preconditions between our work and program verification is that the variable states and properties are on databases. In [4], the authors explored appropriate transaction languages to ensure integrity constraints using weakest preconditions. Calculation of weakest precondition was not discussed. In [11], authors studied the automated construction of artifact-centric workflows so that the workflow execution results are consistent with requirement defined on data, where weakest precondition of each task is calculated.

Rules were given in [20] to derive a set of functional dependencies that hold on query results on given relations. Decidability of dependency implication problem on Datalog programs was studied in [1]. Preservation of integrity constraints by a set of parameterized transactions was studied for relational databases [3] and for semantic databases [26].

7 Conclusion

This paper develops an approach to ensure data integrity within workflow execution by injecting converted constraints into guard of updates. The injection is proved to ensure data integrity while allowing all conservative runs of the original workflow.

The problem raised in this paper is hardly solved, there are many interesting issues to explore further. In one direction, it is desirable to extend this method for constraints with aggregations and arithmetic. Also, conservative requirement for injection can be relaxed by considering more accurate task models such as semantic web services and task contracts in the workflow. Another area of interest is to consider workflow with concurrent executions, which will require new injection techniques. Finally, it is interesting to investigate how techniques such as guard injection can be combined with mechanisms in federated databases [13].

Acknowledgment. The authors are grateful to Richard Hull (IBM T. J. Watson Research Center) for his informative discussions on the GSM semantics.

References

1. Abiteboul, S., Hull, R.: Data functions, datalog and negation. In: Proc. ACM SIGMOD Int. Conf. on Management of Data (1988)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Abiteboul, S., Vianu, V.: A transaction-based approach to relational database specification. Journal of the ACM 36(4), 758–789 (1989)
4. Benedikt, M., Griffin, T., Libkin, L.: Verifiable properties of database transactions. In: Proc. ACM Symposium on Principles of Database Systems (PODS), pp. 117–127 (1996)
5. Bhattacharya, K., Gerede, C., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 288–304. Springer, Heidelberg (2007)

6. Bhattacharya, K., Hull, R., Su, J.: A data-centric design methodology for business processes. In: Handbook of Research on Business Process Modeling. Information Science Publishing (2008)
7. Ceri, S., Widom, J.: Deriving production rules for constraint maintainance. In: Proc. Int. Conf. on Very Large Data Bases (VLDB), pp. 566–577 (1990)
8. Chawathe, S., Garcia-Molina, H., Widom, J.: A toolkit for constraint management in heterogeneous information systems. In: Proc. Int. Conf. on Data Engineering (1996)
9. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 396–412. Springer, Heidelberg (2011)
10. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
11. Fritz, C., Hull, R., Su, J.: Automatic construction of simple artifact-based business processes. In: Proc. Int. Conf. on Database Theory, ICDT (2009)
12. Glushko, R.J., McGrath, T.: Document Engineering: Analyzing and Designing Documents for Business Informatics and Web Services. The MIT Press (2008)
13. Grefen, P., Widom, J.: Protocols for integrity constraint checking in federated databases. Distrib. Parallel Databases 5, 327–355 (1997)
14. Gupta, A., Sagiv, Y., Ullman, J.D., Widom, J.: Constraint checking with partial information. In: Proc. ACM Symp. on Principles of Database Systems (PODS), pp. 45–55 (1994)
15. Gupta, A., Widom, J.: Local verification of global integrity constraints in distributed databases. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 49–58 (1993)
16. Hollingsworth, D.: The workflow reference model: 10 years on. In: Workflow Handbook. Workflow Management Coalition, pp. 295–312 (2004)
17. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath III, F(T.), Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., Vaculin, R.: Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles (Invited talk). In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 1–24. Springer, Heidelberg (2011)
18. Hull, R., Damaggio, E., Masellis, R.D., Fournier, F., Gupta, M., Heath III, F., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., Vaculín, R.: Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events. In: Proc. ACM Int. Conf. on Distributed Event-Based Systems, DEBS (2011)
19. Huyn, N.: Maintaining global integrity constraints in distributed databases. Constraints 2, 377–399 (1997)
20. Klug, A.: Calculating constraints on relational expression. ACM Trans. Database Syst. 5, 260–290 (1980)
21. Künzle, V., Weber, B., Reichert, M.: Object-aware business processes: Fundamental requirements and their support in existing approaches. Int. Journal of Information System Modeling and Design (IJISMD) 2(2), 19–46 (2011)
22. Liu, G., Liu, X., Qin, H., Su, J., Yan, Z., Zhang, L.: Automated realization of business workflow specification. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 96–108. Springer, Heidelberg (2010)
23. Liu, X., Su, J., Yang, J.: Preservation of Integrity Constraints by Workflow: Online Appendix, `http://seg.nju.edu.cn/~liux/pub/CoopIS11_appendix.pdf`
24. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
25. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice-Hall (1992)
26. Su, J.: Dependency preservation in semantic databases. Acta Inf. 31, 27–54 (1994)