Preservation of Integrity Constraints by Workflow: Online Appendix

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

This is an online appendix to our paper [3] (referred to as the conference paper in the following). The online appendix includes the complete formalism for transition system semantics of GSM partially and informally given the conference paper. This semantics is then compared with existing operational semantics of GSM. The proof to the main theorem in the paper the conference paper is also provided. Finally, the constraints and injections of full case study of EzMart is given.

1 Operation schema

The action requesting ASC to send a message m is denoted by $!m$.

Given a GSM workflow model *AP*, the execution model of *AP* is a transition system, denoted by TS_{AP}. The state space is a set of all "snapshots" of artifacts. It is specified by the construct *STATE* in Z notation, where the set of all artifact classes in *AP* is $\mathbf{A} = \{\alpha_i \mid i \in 1 \mathinner{..} n\}$; each artifact class α_i is represented in state space as a table consisting the artifact ID, the data attribute value, and stage and milestone status. *XOp* is the finite set of signatures of possible next operations, where *OPSIG* is the type of operation signatures; *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; *het* and *hct* is the finite set of operations triggered by the head event in *eq* and *cq*, resp.

Full state space schema:

⋆ 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.

STATE

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

Given a stage s and a milestone m, we use $\mathsf{s}.sentry$ to denote the guard sentry of s and $\mathsf{m}^+.sentry$ (resp. $\mathsf{m}^-.sentry$) to denote the achieving (resp. invalidating) sentry of the m, $\mathsf{s}.MS$ to denote the milestones associated with s, $\mathsf{s}.Body$ to denote the defined actions in the body of s, $\mathsf{s}.creatInst : \mathbb{B}$ to indicate whether the stage s is for creating a new artifact instance, $CEv(\mathsf{s}.sentry)$ and $EEv(\mathsf{s}.sentry)$ to denote respectively the set of control event and external event used in $\mathsf{s}.sentry$ (the similar notation applies to milestones), and action *SKIP* to be the short form for $x := x$, denoting no change to any state observables. And given a GSM specification AP, $Ev(AP)$ denotes the set of all events used in the workflow schema.

Given a artifact class α and artifact instance ID aid, $\alpha(aid)$ is short for the selection $\sigma_{id=aid}\alpha$, and $\alpha(aid).x$ denote the projection onto x. Given a stage s or milestone m, s^+, m^+, s^- and m^- denote control event of $\mathsf{s}.open()$, $\mathsf{m}.achieved()$, $\mathsf{s}.closed()$ and $\mathsf{m}.invalidated()$. And $\mathsf{s}^+(\alpha, aid)$ denotes the constructor of the control event with artifact class and artifact ID.

We assume to have function $newid() : ID$ generating a new unique id by some id-generator, and function $sequence(T) : \mathrm{seq}\, T$ takes a finite set T and returns a sequence consisting of all elements in T without repetition with arbitrary order. And in Z notation, symbol $\langle i, t, e, \ldots \rangle$ denotes sequences of items $i, t, e, \ldots$ and $s \frown q$ denotes concatenation of two sequence s and q.

init

$(\alpha_i = \text{ empty table} \mid 1 \leq i \leq n);\ iq = \langle\,\rangle;\ cq = \langle\,\rangle$
$XOp = \{op : Open(\mathsf{s}) \mid \mathsf{s} \in \mathbf{S} \land creatInst(\mathsf{s}) = \text{TRUE}\}$
$het = \varnothing;\ hct = \varnothing$

In TS_{AP}, each operation is identified by its signature which is the operation name and two arguments: stage or milestone name and an optional artifact ID (can omit only for *Open* operation of a create-instance stage, see below). There are three areas in operation specifications, separated by horizontal lines. The first area is for local variable declaration. The second area is the operation guard which is a vertical list of first order formulas. These formulas are connected in conjunction to specify the enabling condition of the transition, i.e. the operation guard. Given an operation op, its guard is denoted by $guard(op)$. The third area is the operation actions for state changes. The changes are given in the form of assignments. Any state variable, artifact attributes and status of stages and milestones not listed in the actions keep unchanged from the current state to the next state. We then explain the six operation types in detail in below.

$Open(\mathsf{s} : \mathbf{S})$

$\alpha : \mathbf{A}$ **where** $\mathsf{s} \in S(\alpha)$
$new : \alpha$ **where**
$\quad new.id = newid() \wedge new.\mathsf{s} = \text{TRUE} \wedge$
$\quad (\forall\, \mathsf{st} : S(\alpha) - \{\mathsf{s}\} \bullet new.\mathsf{st} = \text{FALSE}) \wedge$
$\quad (\forall\, \mathsf{ms} : M(\alpha) \bullet new.\mathsf{ms} = \text{FALSE})$
$cqs =$ **if** $\mathsf{s}^+ \in Ev(AP)$ **then** $\langle \mathsf{s}^+(\alpha, aid) \rangle$
$\quad$ **else** $\langle\,\rangle$

$\mathsf{s}.creatInst = \text{TRUE}$
$Open(s) \in XOp$
$\forall\, id : ID \bullet \alpha(id).\mathsf{s} \notin het \cup hct$
$\forall\, \beta : \mathbf{A};\ id : \beta.ID;\ \mathsf{t} : S(\beta) \bullet$
$\quad \beta(id).\mathsf{t} = \text{FALSE}$
$\mathsf{s}.sentry(\mathbf{A}, eq, cq)$

$\alpha := \alpha \cup \{new\}$
$XOp := (XOp - \{Open(\mathsf{s})\}) \cup$
$\quad \{Body(\mathsf{s}, new.id)\}$
$cq := cq \frown cqs$
$het :=$ **if** $EEv(\mathsf{s}.sentry) = \varnothing$
$\quad$ **then** $het \cup \{new.\mathsf{s}\}$ **else** het
$hct :=$ **if** $CEv(\mathsf{s}.sentry) = \varnothing$
$\quad$ **then** $hct \cup \{new.\mathsf{s}\}$ **else** hct

$Open(\mathsf{s} : \mathbf{S}, aid : ID)$

$\alpha : \mathbf{A}$ **where** $\mathsf{s} \in S(\alpha)$
$mce = \{\, \mathsf{m}^-(\alpha, aid) \mid \mathsf{m} \in \mathsf{s}.MS \wedge$
$\quad \mathsf{m}^- \in Ev(AP) \wedge \alpha(aid).\mathsf{m} = \text{TRUE}\}$
$cqm = sequence(mce)$
$cqs =$ **if** $\mathsf{s}^+ \in Ev(AP)$ **then** $\langle \mathsf{s}^+(\alpha, aid) \rangle$
$\quad$ **else** $\langle\,\rangle$

$\mathsf{s}.creatInst = \text{FALSE}$
$Open(\mathsf{s}, aid) \in XOp$
$\alpha(aid).\mathsf{s} \notin het \cup hct$
$\forall\, \beta : \mathbf{A};\ id : \beta.ID;\ \mathsf{t} : S(\beta) \bullet$
$\quad \beta(id).\mathsf{t} = \text{FALSE}$
$\mathsf{s}.sentry(\mathbf{A}, eq, cq)$

$\alpha(aid).\mathsf{s} := \text{TRUE}$
$\forall\, \mathsf{m} : \mathsf{s}.MS \bullet \alpha(aid).\mathsf{m} := \text{FALSE}$
$XOp := (XOp - \{Open(\mathsf{s})\}) \cup$
$\quad \{Body(\mathsf{s}, aid)\}$
$cq := cq \frown cqs$
$het :=$ **if** $EEv(\mathsf{s}.sentry) = \varnothing$
$\quad$ **then** $het \cup \{\alpha(aid).\mathsf{s}\}$ **else** het
$hct :=$ **if** $CEv(\mathsf{s}.sentry) = \varnothing$
$\quad$ **then** $hct \cup \{\alpha(aid).\mathsf{s}\}$ **else** hct

Fig. 1: Operations: Part 1 – stage open with or without instance creation

Here we fist give the informal description of operations:

- *Open* is responsible to handling stage opening. Given a stage s, the guard of the operation requires: the operation signature is in *XOp*; the same external event or control event can only be used to open each s once; there is only one stage being open at the same time, and the sentry of the stage guard evaluates true under current artifact attribute valuations and events in *eq* and *cq*. When the operation guard is satisfied, the stage is open and all of its milestones are invalidated. Status and control event queues are updated accordingly. And if the stage is triggered by some external or control event heads in the corresponding queue, such stage is added into *het* or *hct*, resp. Also the *Body* of the stage is put in *XOp*, and the current operation is removed from *XOp*.
- *AchieveClose* specifies how a milestone is achieved and thus close the stage. Given a milestone m and its belonging stage s, the operation guard is satisfied when the operation signature is in *XOp*, milestone m and stage s of the artifact is not achieved

$\underline{\mathit{AchieveClose}(\mathsf{m}, \mathit{aid})}$

$\alpha : \mathbf{A}$ **where** $\mathsf{m} \in M(\alpha)$
$\mathsf{s} : S(\alpha)$ **where** $\mathsf{m} \in \mathsf{s}.\mathit{MS}$
$\mathit{cqm} =$ **if** $\mathsf{m}^+ \in \mathit{Ev}(\mathit{AP})$ **then** $\langle \mathsf{m}^+(\alpha, \mathit{aid}) \rangle$ **else** $\langle \, \rangle$
$\mathit{cqs} =$ **if** $\mathsf{s}^- \in \mathit{Ev}(\mathit{AP})$ **then** $\langle \mathsf{s}^-(\alpha, \mathit{aid}) \rangle$ **else** $\langle \, \rangle$

$\mathit{AchieveClose}(\mathsf{m}, \mathit{aid}) \in \mathit{XOp}$
$\alpha(\mathit{aid}).\mathsf{m} = \text{FALSE}$
$\alpha(\mathit{aid}).\mathsf{s} = \text{TRUE}$
$\alpha(\mathit{aid}).\mathsf{m} \notin \mathit{het} \cup \mathit{hct}$
$\mathsf{m}^+.\mathit{sentry}(\mathbf{A}, \mathit{eq}, \mathit{cq})$

$\alpha(\mathit{aid}).\mathsf{m} := \text{TRUE}$
$\alpha(\mathit{aid}).\mathsf{s} := \text{FALSE}$
$\mathit{XOp} := (\mathit{XOp} - \{\mathit{AchieveClose}(\mathsf{m}, \mathit{aid})\}) \cup$
  if $\mathsf{s}.\mathit{creatInst}$ **then** $\{\mathit{Invalid}(\mathsf{m}, \mathit{aid}), \mathit{Open}(\mathsf{s})\}$
  else $\{\mathit{Invalid}(\mathsf{m}, \mathit{aid}), \mathit{Open}(\mathsf{s}, \mathit{aid})\}$
$\mathit{cq} := \mathit{cq} \frown \mathit{cqm} \frown \mathit{cqs}$
if $\mathit{EEv}(\mathsf{m}^+.\mathit{sentry}) \neq \varnothing \wedge$ m uses the event to set x
then $\alpha(\mathit{aid}).x := \mathit{head}\ \mathit{eq}$ **else** *SKIP*
$\mathit{het} :=$ **if** $\mathit{EEv}(\mathsf{m}^+.\mathit{sentry}) = \varnothing$
    then $\mathit{het} \cup \{\alpha(\mathit{aid}).\mathsf{m}\}$ **else** *het*
$\mathit{hct} :=$ **if** $\mathit{CEv}(\mathsf{m}^+.\mathit{sentry}) = \varnothing$
    then $\mathit{hct} \cup \{\alpha(\mathit{aid}).\mathsf{m}\}$ **else** *hct*

$\underline{\mathit{Invalid}(\mathsf{m}, \mathit{aid})}$

$\alpha : \mathbf{A}$ **where** $\mathsf{m} \in M(\alpha)$
$\mathsf{s} : S(\alpha)$ **where** $\mathsf{m} \in \mathsf{s}.\mathit{MS}$
$\mathit{cqm} =$ **if** $\mathsf{m}^- \in \mathit{Ev}(\mathit{AP})$
    then $\langle \mathsf{m}^-(\alpha, \mathit{aid}) \rangle$ **else** $\langle \, \rangle$

$\mathit{Invalid}(\mathsf{m}, \mathit{aid}) \in \mathit{XOp}$
$\alpha(\mathit{aid}).\mathsf{m} = \text{TRUE}$
$\alpha(\mathit{aid}).\mathsf{m} \notin \mathit{het} \cup \mathit{hct}$
$\mathsf{m}^-.\mathit{sentry}(\mathbf{A}, \mathit{eq}, \mathit{cq})$

$\alpha(\mathit{aid}).\mathsf{m} := \text{FALSE}$
$\mathit{XOp} := \mathit{XOp} - \{\mathit{Invalid}(\mathsf{m}, \mathit{aid})\}$
$\mathit{cq} := \mathit{cq} \frown \mathit{cqm}$
$\mathit{het} :=$ **if** $\mathit{EEv}(\mathsf{m}^-.\mathit{sentry}) = \varnothing$
    then $\mathit{het} \cup \{\alpha(\mathit{aid}).\mathsf{m}\}$
    else *het*
$\mathit{hct} :=$ **if** $\mathit{CEv}(\mathsf{m}^-.\mathit{sentry}) = \varnothing$
    then $\mathit{hct} \cup \{\alpha(\mathit{aid}).\mathsf{m}\}$
    else *hct*

Fig. 2: Operations: Part 2 – milestone achieving (with stage close) and invalidating

and open, resp., current external and control events are not used to achieve or invalidate the milestone, and the milestone sentry is evaluated true under current artifacts and events in *eq* and *cq*. When the operation guard is satisfied, m is marked as achieved and s is closed. Event queues and *hct* and *het* are updated accordingly (like in *Open*). The *Open* operation of the belonging stage and *Invalid* operation of this milestone is added to *XOp* and the current operation is removed from *XOp*.

- *Invalid* specifies the operation to invalidate a milestone. Given a milestone m, the operation guard is satisfied when the operation signature is in *XOp*, milestone m is already achieved, current external and control events are not used to achieve or invalidate the milestone, and the milestone sentry is evaluated true under current artifacts and events in *eq* and *cq*. When the operation guard is satisfied, m is marked as *not* achieved. Event queues and *hct* and *het* are updated accordingly (like in *Open*). The current operation is removed from *XOp* and no new operation is added.
- *Body* takes action according to the definition of the stage body. As for the guard, it requires the corresponding stage to be open, and the milestone to be not achieved. Other than following the stage body, this operation also put *AchieveClose* of the milestones of the stage in *XOp* while the current operation is removed.
- Operation *DeEQ* and *DeCQ* specifies how events are removed from *eq* and *cq*, resp. These operations do not require to be exists in *XOp* but can only be enabled when

Body(s, aid)

$\alpha : \mathbf{A}$ **where** $\mathsf{s} \in S(\alpha)$

$Body(\mathsf{s}, aid) \in XOp$
$\alpha(aid).\mathsf{s} = \text{TRUE}$
$\forall\, \mathsf{m} : \mathsf{s}.MS \bullet \neg\, \alpha(aid).\mathsf{m}$

---

if $\mathsf{s}.body$ is sending *msg*
then *msg*! **else** *SKIP*
if $\mathsf{s}.body$ is assigning **x** by **exp**
then $\forall\, x_i : \mathbf{x} \bullet \alpha(aid).x_i = exp_i$
else *SKIP*
$XOp := (XOp - \{Body(\mathsf{s}, aid)\}) \cup$
$\{AchievClose(\mathsf{m}.aid) \mid \mathsf{m} \in \mathsf{s}.MS\}$

DeCQ

$\forall\, op : XOp \bullet \neg\, guard(op)$
$cq \neq \langle\,\rangle$
$hct \neq \varnothing$

$cq := tail\, cq$
$hct := \varnothing$

DeEQ

$\forall\, op : XOp \cup \{DeCQ\} \bullet \neg\, guard(op)$
$eq \neq \langle\,\rangle$

$eq := tail\, eq$
$het := \varnothing$

Fig. 3: Operations: Part 3 – stage body and dequeue

all operations in *XOp* cannot be enabled. Furthermore, the head control event can only be removed when it is used to trigger some stage or milestone; and *DeEQ* can only be enabled when *DeCQ* cannot be enabled.

For operation *Open*, see Figure 1; for *AchieveClose* and *Invalid*, see Figure 2; and for *Body*, *DeEQ* and *DeCQ*, see Figure 3.

2 Comparison with existing semantics

The execution model given in [3] is analogous to the incremental semantics in [2]. Although we do not rigidly follow their semantics using PAC rules and B-steps (see [2]), we follow their intuitions; and a sequence of execution between two immediate external even dequeue operations (*DeEQ*) has the same effect as a macro-B-step (sequence of B-steps between handling two incoming events, see [2]). While [1] and [2] focus on GSM itself and equivalence among different semantic models, we detail the management of event queues, operation enabling conditions (guard) and state updates. In out model, enabling of one operation depends only on the current state. Our model is more suitable in analysis and control of the execution. As a result the transition system introduced in this section can help better in understanding when and where the integrity constraints can be violated, and how to prevent such violation.

Focusing on the the fundamental structure of GSM and problem of ensuring data integrity, we overlook some complex features of GSM models and assume all GSM specifications studied in this paper satisfy the following properties.

1. Only atomic stages are used because only such stages are responsible for task invocation and artifact value update.
2. Each sentry is defined using only the artifact attributes with at most one external and control event. That is, stage and milestone status is not used in sentries. This saves us from building and following dependency graphs as in [2].

3. The event takes its immediate effect (on attribute value) only when it is used as a triggering reply event of a milestone. Triggering event of stages are instead used in the assignments in the stage body.

The execution model introduced can be extended to support the GSM model without these assumptions. But the extension is out of the scope of this paper.

3 Notations in SUB and proof of Theorem 1 in [3]

Some notations used in the function SUB in Section 4.2 of [3] to make it more concise. Here we give their definitions:

- When the stage writes attributes of another artifact, say $\beta(bid)$, then the constraint on $\beta.x$ is only required to hold on artifacts of β reference by attribute $\alpha(aid).bid$. Therefore such reference dependencies should be added into the premises by function *explicitref*(κ) under the following procedure.
  For each $v \in \mathit{WriteSet}(\mathtt{S})$ of the form $\beta_1(\cdots(\beta_{k-1}(\beta_k(bid_k).bid_{k-1}).\cdots).bid_1).x$ where $\beta_k = \alpha$, $bid_k = aid$, and $\beta_1.x \in CA(\kappa)$, add all bid_i and other free attributes of β_i $(1 \le i \le k)$ as $\forall$-quantified variable, and connect $\beta_i(bid_i, bid_{i-1}, \dots)$ in conjunction with the premises (the dots represents all other attributes of β_{i_1}).
- Notation $con[\mathbf{exp}/\alpha(id).\mathbf{x}]$ denotes using each exp in $\mathbf{exp}$ s.t. $x := exp$ (x is in $\mathbf{x}$) appears in the body of the stage to substitute simultaneously in con for
  - every $\forall$-quantified variable y that appears in the column of $\alpha.x$ in the relation atom α identified by id, and remove such y from the $\forall$-quantified variable list of con;
  - every $\exists$-quantified variable z in the column of $\alpha.x$ in the relation atom α identified by id (which must be in the consequent), and remove such z from the $\exists$-quantified variable list of con; and
  - if a constant c appears in the column of $\alpha.x$ in the relation atom α identified by id in the premises (resp. consequent), conjunct $c = exp$ withe premises (resp. consequent).
- Similar notation $con[e/x]$ in the if-clause denotes the substitution of e for x in con, and: if x is an ID, remove it from the $\forall$-quantified variable list of con; if x is an artifact relation, remove all variables appear in it from the $\forall$-quantified variable list of con.

Proof sketch of Theorem 1 in [3].

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

Proof (Sketch) The *soundness* can be proved as following. Because TS_{AP} and $InjTS_{AP}$ share the same initial state and no artifact exists on the initial state, no constraints are violated on the initial state. Assume an arbitrary constraint κ holds on an arbitrary state s of a run ρ' of $InjTS_{AP}$. If there is a transition of *Open* following s, then the injection (w.r.t. stage action) is also satisfied. Therefore, on the next state and the second next state (the state after the stage body operation) in ρ', κ is also satisfied.

The *conservative completeness* is proved by contradiction. Assume there is a conservative run ρ in TS_{AP} that is not a run of $InjTS_{AP}$. Then in case the $s_k = s'_k$ and $t_k = t'_k$ but $s_{k+1} \neq s'_{k+1}$ in runs of TS_{AP} and $InjTS_{AP}$, resp. Because all operations are deterministic, therefore it has to be $s_{k+1} = s'_{k+1}$. In case otherwise, (there is a state s_k in ρ s.t. all of the transitions enabled by the state cannot be enabled by the same state in $InjTS_{AP}$), it is proved that for any stage body of the stage $\mathtt{S}$ that t_k is open stage operation of $\mathtt{S}$, if t_k cannot be enabled in $InjTS_{AP}$ by s_k, ρ is either not sound or uses the reply event to update critical variables. In both cases, contradiction are witnessed. Therefore, conservative completeness is proved. □

Proof

Safeness We prove injection safeness by induction. Let $\rho' = s'_0\, t'_0\, \ldots\, t'_{n-1}\, s'_n$ be a complete run of $InjTS_{AP}$, and without loosing generality, consider an arbitrary κ in $\mathbf{K}$. Because in the initial state of $InjTS_{AP}$ is the same as the initial state of TS_{AP}, the set of artifacts are empty, and in Equation (1) in [3], there is at least one artifact relation atoms in the premises of κ, therefore κ holds on s'_0. As induction hypothesis, suppose κ holds on state s'_k ($0 \leq k < n$). Only operations *Body* of some stage $\mathtt{S}$ and *AchievClose* of some milestone $\mathtt{m}$ can change the attribute value. If t'_k is not one of these two operations, then all constraints also hold on s_{k+1}. In case that t'_k is a *Body* operation of some stage $\mathtt{S}$, since there can be at most one stage being active, $\mathtt{SUB}(\kappa, \mathtt{S}\,[, aid])$ must hold on s_k, then after the assignment of $\mathtt{S}.Body$, κ also hold on s'_{k+1}. If t'_k is an *AchievClose* operation, then by Algorithm 1 in [3], it does not change the value of attributes in $CA(\kappa)$ (otherwise, there will be no such t'_k), therefore, κ also holds on s'_{k+1}. Therefore, safeness is established.

Completeness Assume there is a conservative run ρ that has no identical run ρ' in $InjTS_{AP}$, we prove contradiction. Because runs of TS_{AP} and $InjTS_{AP}$ share the same initial state, without loosing generality, suppose run ρ' is the run of $InjTS_{AP}$ that shares the longest common prefix with ρ, and let the maximal common prefix be $s_0 t_0 \ldots t_{k-1} s_k$, where $k \geq 0$. Then there are the following cases for the operations and states following s_k:

1. either there is no t'_k that can be enabled on s_k, or for any operations t'_k enabled by s_k in $InjTS_{AP}$, t'_k and t_k are not of the same operation (with and without injection resp.): Because the injection is only made on *Open* operations, then such operation of t_k must be an $Open(\mathtt{S}\,[, aid])$ operation for some stage $\mathtt{S}$ and optionally artifact id *aid*. Thus,
   - if the body of $\mathtt{S}$ is assignments, because ρ is safe, then t_{k+1} is operation $Body(\mathtt{S}, aid)$, and constraints in $\mathbf{K}$ hold on state s_{k+1}. The injection on $\mathtt{S}$, $Inj(\mathtt{S})$, is a conjunction of $\mathtt{SUB}(\kappa, s\,[, aid])$ for every constraint in $\mathbf{K}$, then $guard(t_k) \wedge Inj(\mathtt{S})$ holds on s_k, which leads to a contradiction that $Open(\mathtt{S}\,[, aid])$ with injection is enabled on s_k in $InjTS_{AP}$;
   - if the body of $\mathtt{S}$ is sending a one-way invocation or a reply, then operation $Open(\mathtt{S}\,[, aid])$ is not injected, and therefore $Open(\mathtt{S}\,[, aid])$ with injection is enabled on s_k in $InjTS_{AP}$;
   - if the body of $\mathtt{S}$ is sending a two-way invocation, there must be a milestone triggered by the response. In case the milestone dose not use the response to set

any attributes in $CA(\kappa)$ for any constraint $\kappa \in \mathbf{K}$, then $Open(\mathsf{s}\,[, aid])$ operation is not injected and thus is enabled on state s_k. Otherwise, it is injected with FALSE and cannot be enabled. Thus ρ cannot be conservative.

Therefore, we witness contradictions.

2. t'_k and t_k are of the same operation (with and without injection resp.) but $s'_{k+1} \neq s_{k+1}$: Because ρ and ρ' share s_k, and all operations produces deterministic effect, then s_k must be the same of s'_k. We witness a contradiction.

In all of the cases we witness a contradiction. Therefore, for each conservative run ρ of TS_{AP}, there is be a complete run ρ' of $InjTS_{AP}$ s.t. ρ and ρ' are identical.

□

4 Constraints and injection on EzMart

In this appendix section, we give the constraint formulas and the injection. Note that in the substitutions, $id \circ newid()$ is further replaced by FALSE for any id unless $\circ$ is $\neq$.; and $x \circ \mathsf{null}$ is further replaced by FALSE for any unless x is null or $\circ$ is $\neq$. Except these trivial replacements, no reduction is made on the injection.

Attribute constraint

In Customer:

$$\forall\, custid, email, \ldots\, .\, Customer(custid, email) \rightarrow email \neq \text{`'}$$

The injection on $Open(\mathsf{register})$ is

$$\text{TRUE} \rightarrow \mathtt{regreq}.email \neq \text{`'}$$

In Order:

$$\forall\, ordid, custid, invid, qty, \ldots\, . \\ Order(ordid, custid, invid, \ldots, qty, \ldots) \rightarrow custid \neq \mathsf{null} \wedge invid \neq \mathsf{null} \wedge qty > 0$$

The injection on $Open(\mathsf{create})$ is

$$\forall\, ordid, custid, invid, qty, \ldots\, .\, \text{TRUE} \rightarrow \\ (head\ cq).custid \neq \mathsf{null} \wedge (head\ cq).invid \neq \mathsf{null} \wedge (head\ cq).qty > 0$$

In Ship:

$$\forall\, shipid, ordid, addr, name, from, ship\_stat\,.\; Ship(shipid, ordid, addr, name, from, ship\_stat) \rightarrow ordid \neq \mathsf{null} \wedge addr \neq \mathsf{null} \wedge name \neq \mathsf{null} \wedge from \neq \mathsf{null} \wedge addr \neq from$$

The injection on $Open(\mathsf{prepare})$ is

$$\forall\, shipid, ordid, addr, name, from, ship\_stat\,.\; \text{TRUE} \rightarrow (head\ cq).ordid \neq \mathsf{null} \wedge Cutomer(Order((head\ cq).ordid).custid).addr \neq \mathsf{null} \wedge Cutomer(Order((head\ cq).ordid).custid).name \neq \mathsf{null} \wedge Inventory(Order((head\ cq).ordid).invid).loc \neq \mathsf{null} \wedge Cutomer(Order((head\ cq).ordid).custid).addr \neq Inventory(Order((head\ cq).ordid).invid).loc$$

where $head\ cq$ is a control event of paid^{+}.

In Inventory:

$$\forall\, invid, loc, \ldots\,.\; Inventory(invid, \ldots, avail\_qty, \ldots) \rightarrow loc \neq \mathsf{null}$$

The injection on $Open(\mathsf{inv\_initiate})$ is

$$\text{TRUE} \rightarrow (head\ eq).loc \neq \mathsf{null}$$

where $head\ eq$ is a `invInit` event.

Candidate key constraint

The constraint formula is:

$$\forall\, custid, email, custid', \ldots\,.\; Customer(custid, email, \ldots) \wedge Customer(custid', email, \ldots) \rightarrow custid = custid'$$

The injection on $Open(\mathsf{register})$ is

$$\forall\, custid', \ldots\,.\; Customer(custid', (head\ eq).email, \ldots) \rightarrow \text{FALSE}$$

where $head\ eq$ is a `regreq` event.

Foreign key constraint

There is foreign key reference from *Order* to *Customer*, *Order* to *Ship*, *Order* to *Inventory* and *Ship* to *Order*.

In Order:

$$\forall\, ordid, custid, invid, \ldots\,.\; Order(ordid, custid, invid, \ldots) \rightarrow \exists \ldots\,.\; Customer(custid, \ldots) \wedge Inventory(invid, \ldots)$$

$$\forall\, ordid, shipid, \ldots\,.\; Order(ordid, \ldots, shipid, \ldots) \wedge shipid \neq \mathsf{null} \rightarrow \exists \ldots\,.\; Ship(shipid, \ldots)$$

The injection on $\mathit{Open}(\mathsf{create})$ is

$$\text{TRUE} \rightarrow \exists \ldots \centerdot \mathit{Customer}((\mathit{head}\ \mathit{eq}).\mathit{custid}, \ldots) \wedge \mathit{Inventory}((\mathit{head}\ \mathit{eq}).\mathit{invid}, \ldots)$$

where *head eq* is a `checkout` event.

The injection on $\mathit{Open}(\mathsf{ship}, \mathit{shipid})$ is (*shipid* of *Order* is set in ship of *Ship*)

$$\forall \mathit{ordid}, \ldots \centerdot \mathit{Ship}(\mathit{shipid}, \mathit{ordid}, \ldots) \wedge \mathit{Order}(\mathit{ordid}, \mathit{shipid}, \ldots) \wedge \mathit{shipid} \neq \mathsf{null} \rightarrow \exists \ldots \centerdot \mathit{Ship}(\mathit{shipid}, \ldots)$$

Obviously, this results in TRUE.

In Ship:

$$\forall \mathit{shipid}, \mathit{ordid}, \ldots \centerdot \mathit{Ship}(\mathit{shipid}, \mathit{ordid}, \ldots) \rightarrow \exists \ldots \centerdot \mathit{Order}(\mathit{ordid}, \ldots)$$

The injection on $\mathit{Open}(\mathsf{prepare})$ is

$$\text{TRUE} \rightarrow \exists \ldots \centerdot \mathit{Order}((\mathit{head}\ \mathit{cq}).\mathit{getid}(), \ldots)$$

where *head cq* is a paid^{+} event.

Ship-order reference circle

It is actually two constraints:

$$\forall \mathit{ordid}_1, \mathit{ordid}_2, \mathit{shipid}, \ldots \centerdot \mathit{Order}(\mathit{ordid}_1, \ldots, \mathit{shipid}, \ldots) \wedge \mathit{Ship}(\mathit{shipid}, \mathit{ordid}_2, \ldots) \rightarrow \mathit{ordid}_1 = \mathit{ordid}_2$$
$$\forall \mathit{shipid}_1, \mathit{shipid}_2, \mathit{ordid}, \ldots \centerdot \mathit{Order}(\mathit{ordid}, \ldots, \mathit{shipid}_1, \ldots) \wedge \mathit{Ship}(\mathit{shipid}_2, \mathit{ordid}, \ldots) \rightarrow \mathit{shipid}_1 = \mathit{shipid}_2$$

The injection on $\mathit{Open}(\mathsf{create})$ is

$$\forall \mathit{ordid}_2, \ldots \centerdot \mathit{Ship}(\mathsf{null}, \mathit{ordid}_2, \ldots) \rightarrow \text{FALSE}$$
$$\forall \mathit{shipid}_2 \centerdot \text{FALSE} \rightarrow \mathsf{null} = \mathit{shipid}_2$$

The injection on $\mathit{Open}(\mathsf{prepare})$ is

$$\forall \mathit{ordid}_1, \mathit{shipid}, \ldots \centerdot \mathit{Order}(\mathit{ordid}_1, \ldots, \mathit{shipid}, \ldots) \rightarrow \mathit{ordid}_1 = (\mathit{head}\ \mathit{cq}).\mathit{getid}()$$
$$\forall \mathit{shipid}_1, \mathit{shipid}_2, \mathit{ordid}, \ldots \centerdot \mathit{Order}(\mathit{ordid}, \ldots, \mathit{shipid}_1, \ldots) \rightarrow \text{FALSE}$$

where *head cq* is a paid^{+} event.

The injection on $\mathit{Open}(\mathsf{ship}, \mathit{shipid})$ is

$$\forall \mathit{ordid}_1, \mathit{ordid}_2, \mathit{shipid}, \ldots \centerdot \mathit{Order}(\mathit{ordid}_1, \ldots, \mathit{shipid}, \ldots) \wedge \mathit{Ship}(\mathit{shipid}, \mathit{ordid}_2, \ldots) \wedge \mathit{Ship}(\mathit{shipid}, \mathit{ordid}_1, \ldots) \rightarrow \mathit{ordid}_1 = \mathit{ordid}_2$$
$$\forall \mathit{shipid}_1, \mathit{shipid}_2, \mathit{ordid}, \ldots \centerdot \mathit{Order}(\mathit{ordid}, \ldots, \mathit{shipid}_1, \ldots) \wedge \mathit{Ship}(\mathit{shipid}, \mathit{ordid}, \ldots) \rightarrow \mathit{shipid}_1 = \mathit{shipid}$$

Address-name constraint

The constraint is

$$\forall\, ordid, custid, shipid, addr_c, name_c, addr_s, name_s \ldots \,.\, Order(ordid, custid, shipid, \ldots) \land \\ Customer(custid, \ldots, addr_c, name_c, \ldots) \land Ship(shipid, ordid, addr_s, name_s) \rightarrow \\ addr_c = addr_s \land name_c = name_s$$

The injection on *Open*register is

$$\forall\, shipid, addr_s, name_s \ldots \,.\, \text{FALSE} \land Ship(shipid, ordid, addr_s, name_s) \rightarrow \\ \texttt{regreq}.addr = addr_s \land \texttt{regreq}.name = name_s$$

The injection on *Open*(create) is

$$\forall\, addr_c, name_c, addr_s, name_s \ldots \,. \\ Customer((head\ cq).custid, \ldots, addr_c, name_c, \ldots) \land \text{FALSE} \rightarrow \\ addr_c = addr_s \land name_c = name_s$$

The injection on *Open*(prepare) is

$$\forall\, ordid, custid, addr_c, name_c, \ldots \,.\, \text{FALSE} \land \\ Customer(custid, ordid, addr_c, name_c, \ldots) \rightarrow \\ addr_c = Customer(Order((head\ cq).getid()).custid).addr \land \\ name_c = Customer(Order((head\ cq).getid()).custid).name$$

The injection on *Open*(ship, *shipid*) is

$$\forall\, ordid, custid, addr_c, name_c, addr_s, name_s \ldots \,.\, Order(ordid, custid, shipid, \ldots) \land \\ Customer(custid, \ldots, addr_c, name_c, \ldots) \land Ship(shipid, ordid, addr_s, name_s) \rightarrow \\ addr_c = addr_s \land name_c = name_s$$

Ship-from constraint

The constraint formula is

$$\forall\, ordid, shipid, invid, loc, from, \ldots \,.\, Order(ordid, \ldots, invid, shipid, \ldots) \land \\ Ship(shipid, \ldots, from, \ldots) \land Inventory(invid, \ldots, loc, \ldots) \rightarrow \\ loc = from$$

The injection on *Open*(create) is

$$\forall\, from, loc \ldots \,. \\ \text{FALSE} \land Inventory((head\ cq).invid, \ldots, loc, \ldots) \rightarrow \\ loc = from$$

The injection on *Open*(prepare) is

$$\forall\, ordid, invid, loc, \ldots \,.\, \text{FALSE} \land \\ Ship(shipid, \ldots, Inventorty(Order((head\ cq).getid()).invid).loc, \ldots) \land \\ Inventory(invid, \ldots, loc, \ldots) \rightarrow \\ loc = Inventorty(Order((head\ cq).getid()).invid).loc$$

The injection on $Open(\textsf{ship}, shipid)$ is

$$\forall\, ordid, invid, loc, from, \ldots \,.\, Order(ordid, \ldots, invid, shipid, \ldots) \wedge$$
$$Ship(shipid, ordid, from, \ldots) \wedge Inventory(invid, \ldots, loc, \ldots) \rightarrow$$
$$loc = from$$

Status constraint

The constraint formula is

$$\forall\, ordid, shipid, ord\_stat, ship\_stat \,.\, Order(ordid, \ldots, shipid, \ldots, ord\_stat) \wedge$$
$$Ship(shipid, ordid, \ldots, ship\_stat) \wedge ship\_stat \neq \text{FINISH} \wedge ship\_stat \neq \text{FAILED} \rightarrow$$
$$ord\_stat \neq \text{RETURN} \wedge ord\_stat \neq \text{CANCEL}$$

The injection on $Open(\textsf{create})$ is

$$\forall\, ship\_stat \,.\, \textsc{false} \wedge ship\_stat \neq \text{FINISH} \wedge ship\_stat \neq \text{FAILED} \rightarrow$$
$$\text{CREATE} \neq \text{RETURN} \wedge \text{CREATE} \neq \text{CANCEL}$$

The injection on $Open(\textsf{prepare})$ is

$$\forall\, ordid, ord\_stat, ship\_stat \,.\, \textsc{false} \wedge$$
$$\text{PREPAR} \neq \text{FINISH} \wedge \text{PREPAR} \neq \text{FAILED} \rightarrow$$
$$ord\_stat \neq \text{RETURN} \wedge ord\_stat \neq \text{CANCEL}$$

The injection on $Open(\textsf{ship}, shipid)$ is

$$\forall\, ord\_stat, ship\_stat \,.\, Order(ordid, \ldots, shipid, \ldots, ord\_stat) \wedge$$
$$Ship(shipid, ordid, \ldots, ship\_stat) \wedge \text{SHIPIN} \neq \text{FINISH} \wedge \text{SHIPIN} \neq \text{FAILED} \rightarrow$$
$$ord\_stat \neq \text{RETURN} \wedge ord\_stat \neq \text{CANCEL}$$

The injection on $Open(\textsf{sell}, invid)$ is

$$\forall\, ordid, shipid, ship\_stat \,.\, Order(ordid, \ldots, invid, , shipid, \ldots, ord\_stat) \wedge$$
$$Ship(shipid, ordid, \ldots, ship\_stat) \wedge ship\_stat \neq \text{FINISH} \wedge ship\_stat \neq \text{FAILED} \wedge$$
$$Inventory(invid, \ldots) \rightarrow$$
$$\text{INVUPD} \neq \text{RETURN} \wedge \text{INVUPD} \neq \text{CANCEL}$$

The injection on $Open(\textsf{further\_action})$ is $\textsc{false}$.

References

1. E. Damaggio, R. Hull, and R. Vaculin. On the equivalence of incremental and fixpoint semantics for business entities with guard-stage-milestone lifecycles. In *Proc. Int. Conf. on Business Process Management (BPM)*, 2011.
2. R. Hull, E. Damaggio, R. D. Masellis, F. Fournier, M. Gupta, F. Heath III, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculín. Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events. In *Proc. ACM Int. Conf. on Distributed Event-Based Systems (DEBS)*, 2011.
3. X. Liu, J. Su, and J. Yang. Preservation of Integrity Constraints by Workflow. In *Proc. Int. Conf. on Cooperative Information Systems (CoopIS)*, 2011. (to appear).