Formalizing Guard-Stage-Milestone meta-models as Data-Centric Dynamic Systems

Dmitry Solomakhin[1], Marco Montali[1], and Sergio Tessaris[1]

Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy
(solomakhin|montali|tessaris)@inf.unibz.it

Table of Contents

1 Introduction

In the last decade, a plethora of graphical notations (such as BPMN and EPCs) have been proposed to capture business processes. Independently from the specific notation at hand, formal verification has been generally considered as a fundamental tool in the process design phase, supporting the modeler in building correct and trustworthy process models [11]. Intuitively, formal verification amounts to check whether possible executions of the business process model satisfy some desired properties, like generic correctness criteria (such as deadlock freedom or executability of activities) or domain-dependent constraints. To enable formal verification and other forms of reasoning support, the business process language gets translated into a corresponding formal representation, which typically relies on variants of Petri nets [15], transition systems [1], or process algebras [14]. Properties are then formalized using temporal logics, using model checking techniques to actually carry out verification tasks [4].

A common drawback of classical process modeling approaches is being *activity-centric*: they mainly focus on the control-flow perspective, lacking the connection between the process and the data manipulated during its executions. This reflects also in the corresponding verification techniques, which often abstract away from the data component. This "data and process engineering divide" affects many contemporary process-aware information systems, incrementing the amount of redundancies and potential errors in the development phase [8]. To tackle this problem, the artifact-centric paradigm has recently emerged as an approach in which processes are guided by the evolution of business data objects, called *artifacts* [12,5]. A key aspect of artifacts is coupling the representation of data of interest, called *information model*, with *lifecycle constraints*, which specify the acceptable evolutions of the data maintained by the information model. On the one hand, new modeling notations are being proposed to tackle artifact-centric

processes. A notable example is the Guard-State-Milestone (GSM) graphical notation [6], which corresponds to way executive-level stakeholders conceptualize their processes [3]. On the other hand, formal foundations of the artifact-centric paradigm are being investigated in order to capture the relationship between processes and data and support formal verification [7,2,9]. Two important issues arise in this setting. First, verification formalisms must go beyond propositional temporal logics, and incorporate first-order formulae to express constraints about the evolution of data and to query the information model of artifacts. Second, formal verification becomes much more difficult than for classical activity-centric approaches, even undecidable in the general case.

In this technical report, we tackle the problem of *automated verification of GSM models*. Our long-term goal is to provide support during the design of a GSM process, assisting the modeler in checking whether the process satisfies some desired properties. As an underlying formalism, we consider the framework of Data-Centric Dynamic Systems (DCDSs) [9]. DCDSs are systems composed of a data layer manipulated by a process layer, which can interact with external services in order to inject new, fresh information into the data layer. These services can represent internal components treated as a black box (such as the component responsible to assign an identifier to a newly created artifact instance), or truly external partners (such as the environment which a GSM model can interact with). Several decidability results concerning the verification of DCDSs have been recently established, considering the two settings in which services behave deterministically and non-deterministically, and investigating different fragments of first-order μ-calculus [13] as verification languages. More specifically, we study how to formalize GSM by relying on its *incremental semantics*, one of the three equivalent GSM execution semantics introduced in [6]. The possibility of translating a GSM model into a corresponding DCDS is the basis for applying the decidability results and verification techniques discussed in [9] for the abstract DCDS approach, to the concrete case of GSM.

The main contribution of the present report is to provide technical details on a formalization procedure for GSM using data-centric dynamic system (DCDS).

2 Overview of data-centric dynamic systems

Despite having a formally specified operational semantics for GSM models [6], the verification of different properties of such models (e.g. existence of complete execution, safety properties) is still an open problem. In order to solve this problem, one should define a particular formalism that captures the intended operational semantics of the business artifacts and provides mechanisms to solve different verification tasks.

One of the most promising candidates for such a formalism is a data-centric dynamic system (DCDS) together with its general verification framework presented in [10]. A DCDS is a pair $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, where $\mathcal{D}$ is a data layer and $\mathcal{P}$ is a process layer over the former.

The data layer $\mathcal{D}$ models the relevant database schema together with its set of integrity constraints, while the process layer $\mathcal{P}$ is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where

- $\mathcal{F}$ is a finite set of functions representing interfaces to external services.
- $\mathcal{A}$ is a set of actions of the form $\alpha(p_1, ..., p_n) : \{e_1, ..., e_m\}$, where $p_1, ..., p_n$ are input parameters of an action and e_i are effects of an action. Each effect e_i has the form $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$, where
  - q_i^+ is a union of conjunctive queries (UCQ) over $\mathcal{D}$ that select the tuples to instantiate the effect.
  - Q_i^- is an arbitrary FO formula that filters away some tuples obtained by q_i^+.
  - E_i is the effect, i.e. a set of generated facts for $\mathcal{D}$.
- ϱ is a process which is a finite set of condition-action rules of the form $Q \mapsto \alpha$, where α is an action and Q is a FO query over $\mathcal{D}$.

The execution semantics of a DCDS $\mathcal{S}$ is defined by a possibly infinite-state transition system $\Upsilon_{\mathcal{S}}$, where states are instances of the database schema in $\mathcal{D}$ and each transition corresponds to the application of an executable action in $\mathcal{P}$. In DCDSs the source of infinity relies in the service calls, which can inject arbitrary fresh values into the system.

3 From GSM artifact model to DCDS

First of all, we need to represent the data layer of the GSM model R in a DCDS system $\mathcal{S}_R = \langle \mathcal{D}, \mathcal{P} \rangle$.

3.1 Data layer

Given an artifact type $(R, x, Att_{data} \cup Att_{status}, Typ, Stg, Mst, Lcyc)$,
a lifecycle $Lcyc = (Substages, Task, Owns, Guards, Ach, Inv)$[1],
a set of finite sets **MSG** of message types and **SRV** of (2-way) services,
a corresponding set of PAC-rules Γ_{PAC};
the corresponding data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_\prime \rangle$ will have the following form:

- $\mathcal{C} = \bigcup_i \mathbf{DOM}(Typ(Att_i))$,
- $\mathcal{R} = \{R_{att}\} \cup \{R_{chg}^{m_i} | m_i \in Mst\} \cup \{R_{chg}^{s_j} | s_j \in Stg\} \cup$
  $\cup \ \{R_{data}^{msg_k} | msg_k \in \mathbf{MSG}$ and msg_k is incoming 1-way message$\} \cup$
  $\cup \ \{R_{data}^{srv_p} | srv_p \in \mathbf{SRV}$ and srv_p is a service call return$\} \cup$
  $\cup \ \{R_{out}^{msg_q} | msg_q \in \mathbf{MSG}$ and msg_q is outgoing 1-way message$\} \cup$
  $\cup \ \{R_{exec}, R_{block}\}$,
  where:

[1] wlog, for the sake of simplicity we consider $Substages = Inv = \emptyset$

- $R_{att} = (id_A, fr, a_1, ..., a_n, s_1, ..., s_m, m_1, ..., m_k)$, where id_A ranges along the artifact IDs, $n = |Att_{data}|, m = |Stg|, k = |Mst|$. Stores the attributes of an artifact. Each s_i and m_i store the status of a stage and milestone, respectively.
- $R_{chg}^{m_i} = (id_R^{origin}, newstate)$ – stores the fact of a milestone m_i has been recently achieved or invalidated; id_R^{origin} is the id of the artifact where it happened and $newstate$ stores the new value. This relation is used to model the pool of internal events concerning milestones.
- $R_{chg}^{s_j} = (id_R^{origin}, newstate)$ – same as previous but about stages being opened or closed.
- $R_{data}^{msg_k} = (id_R^{dest}, p_1, ..., p_l)$, where msg_k is a 1-way incoming message from the environment, $(p_1, ..., p_l)$ – its signature and id_R^{dest} is the id of a destination artifact (or $null$ if message is indirected)[2].
  Used to model the immediate effect of an incoming message.The idea behind it is that, together with propagating changes of involved attributes, message is passed to the inner 'data-pool' so that all the sentries which use this message in the event expression, could react properly.
- $R_{data}^{srv_p} = (id_R^{caller}), p_1, ..., p_l$, where srv_p is an external service call return, $(p_1, ..., p_l)$ – its signature and id_R^{dest} is the id of a caller artifact (basicaly the id of a destination artifact for service call return, but it can't be $null$). **Same as for the incoming message**.
- $R_{out}^{msg_q} = (id_R^{dest}, a_1, ..., a_l)$ – stores eventual outgoing messages to be sent to the environment after finishing the B-step.
- $R_{exec} = (id_R, x_1, ..., x_c)$, where x_i are flags that keep information on which PAC rules have been taken in consideration and $c = |\Gamma_{PAC}|$.
- $R_{block} = (id_R, blocked)$, keeps information whether an artifact instance may receive an incoming message / service call return, or is currently still processing the previous one.
- $SHELF = (index, id_R)$, implements a proposed methodology of keeping the ACS data-bounded by restricting the number of artifact instances within one snapshot (the number of instances along the execution path may still be infinite).
  Represents a physical storage for artifact instances, where one shelf may contain only one artifact instance. When there is no artifact instance on the shelf - the ID of stored instance is -1.

– $\mathcal{E} = \{\mathcal{E}_i | \mathcal{E}_i$ is some integrity constraint$\}$
– $\mathcal{I}_0 = \emptyset$.

3.2 Process layer

Given an artifact type $(R, x, Att_{data} \cup Att_{status}, Typ, Stg, Mst, Lcyc)$,
a lifecycle $Lcyc = (Substages, Task, Owns, Guards, Ach, Inv)$,
a set of finite sets **MSG** of message types and **SRV** of (2-way) services,
a corresponding set of PAC-rules Γ_{PAC};
the corresponding process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ will have the following form:

[2] Q: Broadcast messages? A: They use queries to address specific artifacts.

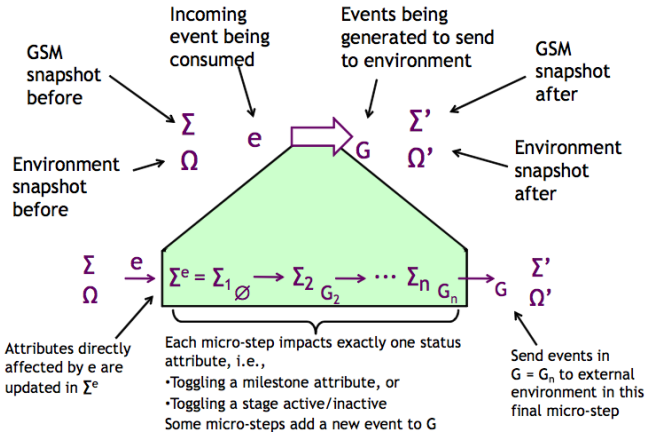

Fig. 1. Incremental formulation of a B-step [6]

- $\mathcal{F} = \{f^{genID}\} \cup \{f^{srv_i} | srv_i \in \mathbf{SRV}\} \cup$
  $\cup \{f^{msg_i} | msg_i \in \mathbf{MSG}$ and msg_i is 1-way message from environment$\} \cup$
  $\cup \{f^{msg_i}_{out} | msg_i \in \mathbf{MSG}$ and msg_i is 1-way outgoing message$\}$, where
  - f^{genID} is a function with generates IDs for newly created artifact instances.
  - $srv_i(\overline{x}) = (f^{srv_i}(\overline{x}, 1), ..., f^{srv_i}(\overline{x}, n))$, where n is cardinality of service output signature.
  - $msg_i = (f^{msg_i}(1), ..., f^{msg_i}(n))$, where n is cardinality of message signature.
- $\mathcal{A} = \{\alpha_i\}$ is a set of actions, where $i = \overrightarrow{1, ..., N_A}$ and
  $N_A = |\Gamma_{PAC}| + |MSG_{in}| + |SRV| + 1 + 1 + 1$, i.e. there is
  - one action for every PAC-rule of the given GSM model
  - one action for each incoming message (to describe immediate effect);
  - one action for each service call (for immediate effect of the call return);
  - one action to send outgoing messages after each B-step;
  - one action to create an artifact instance;
  - one action to remove the artifact;
- A process ϱ, which is a set of condition-action rules is described below, where for each action $\alpha_i \in \mathcal{A}$ there is one and only condition-action rule defined.

B-step in DCDS When modeling a GSM model as a DCDS system, what we would like to do is to mimic an incremental semantics of GSM, i.e. we encode each micro-step of the B-step as a separate condition-action rule in DCDS system, such that the effect on the data and process layer of the ACS of this action coincides with the effect of corresponding micro-step in GSM.

Recall the structure of a B-step in GSM represented on Figure 1. According to the incremental formulation of GSM, each B-step consists of an initial micro-

step which incorporates incoming event into current snapshot, a sequence of micro-steps executing all applicable PAC-rules, and finally a micro-step sending a set of generated events at the termination of the B-step. The translation relies on the incremental semantics: given a GSM model $\mathcal{G}$, we encode each possible micro-step as a separate condition-action rule in the process of a corresponding DCDS system $\mathcal{S}$, such that the effect on the data and process layers of the action coincides with the effect of the corresponding micro-step in GSM. However, in order to guarantee that the transition system induced by a resulting DCDS mimics the one of the GSM model, the translation procedure should also ensure that all semantic assumption of GSM are modeled properly: (i) "one-message-at-a-time" and "toggle-once" principles, (ii) the finiteness of micro-steps within a B-step, and (iii) their order imposed by the model. We sustain these requirements by introducing into the data layer a set of auxiliary relations, suitably recalling them in the CA-rules to reconstruct the desired behaviour.

Thus, when performing the translation we rely on the following assumptions:

1. Restricting $\mathcal{S}$ to process only one incoming message at a time is implemented by the introduction of a *blocking mechanism*, represented by an auxiliary relation $R_{block}(id_R, blocked)$ for each artifact in the system, where id_R is the artifact instance identifier, and *blocked* is a boolean flag. This flag is set to *true* upon receiving an incoming message, and is then reset to $false$ at the termination of the corresponding B-step, once the outgoing events accumulated in the B-step are sent the environment. If an artifact instance has $blocked = true$, no further incoming event will be processed. This is enforced by checking the flag in the condition of each CA-rule associated to the artifact.
2. In order to ensure "toggle once" principle and guarantee the finiteness of sequence of micro-steps triggered by an incoming event, we introduce an *eligibility tracking mechanism*. This mechanism is represented by an auxiliary relation $R_{exec}(id_R, x_1, ..., x_c)$, where c is the total number of PAC-rules, and each x_i corresponds to a certain PAC-rule of the GSM model. Each x_i encodes whether the corresponding PAC rule is eligible to fire at a given moment in time (i.e., a particular micro-step). The initial setup of the eligibility tracking flags is performed at the beginning of a B-step, based on the evaluation of the prerequisite condition of each PAC rule. More specifically, when $x_i = 0$, the corresponding CA-rule is eligible to apply and has not yet been considered for application. When instead $x_i = 1$, then either the rule has been fired, or its prerequisite turned out to be false.
3. The same flag-based approach is used to propagate in a compact way information related to the PAC rules that have been already processed, following a mechanism that resembles *dead path elimination* in BPEL. In fact, R_{exec} is also used to enforce a firing order of CA-rules that follows the one induced by $\mathcal{G}$. This is achieved as follows. For each CA-rule $Q \mapsto \alpha$ corresponding to a given PAC rule r, condition Q is put in conjunction with a further formula, used to check whether all the PAC rules that precede r according to

the ordering imposed by $\mathcal{G}$ have been already processed. Only in this case r can be considered for application, consequently applying its effect α to the current artifact snapshot. More specifically, the corresponding CA-rule becomes $Q \wedge exec(r) \mapsto \alpha$, where $exec(r) = \bigwedge_i x_i$ such that i ranges over the indexes of those rules that precede r. Once all x_i flags are switched to 1, the B-step is about to finish: a dedicated CA-rule is enabled to send the outgoing events to the environment, and the artifact instance *blocked* flag is released.

So, here is the general algorithm of translation:

1. For each incoming message m_i, construct a CA-rule, which:
   - Implements immediate effect of an incoming message
   - Puts a block on the artifact instance to perform the B-step.
   - Sets up the eligibility flags based on the current snapshot, i.e. for each PAC-rule check the prerequisite part. If $\pi = false$, then set the boolean flag of the corresponding micro-step 1 (which will basically mean that we have already considered this rule).
2. For each PAC-rule r_i construct a CA-rule such that:
   - It contains a check whether the action has been already executed or has been marked as irrelevant (simply checks the boolean flag).
   - It contains a check whether all the PAC rules that precede r_i according to the ordering imposed by $\mathcal{G}$ have been already processed.
   - If relevant and eligible and if the antecedent part is true (the query to populate the effect is not empty), performs the required change of the status attribute
   - If relevant and eligible, marks the corresponding boolean flag as $true$.
3. Construct a CA-rule, which will send outgoing messages and unblock the artifact instance:
   - Check whether all the PAC-rules have been taken into consideration ($\forall x_k \in R_{block} : x_k = 1$) and whether the artifact instance is still blocked.
   - In the action part – release the block.
   - In the action part – flush the eligibility flags.
4. If *create artifact* or *remove artifact* tasks are present, add micro-steps dealing with it (a particular type of the immediate effect micro-steps described later).

Now let us get down to translating each of the possible micro-steps.

Translation 1 (Immediate effect of 1-way incoming message).
Assume an incoming message type M, its associated artifact type R and its signature $(a_1 : Typ(a_1), ...a_k : Typ(a_k))$, where $a_i \in Att_{data}$.
Assume also a set of PAC-rules $\{(\pi_i, \alpha_i, \gamma_i)\}$.
Then an immediate effect of a message of type M on some artifact instance A

of type R may be modeled by the following condition-action rule:

$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto$

$\alpha_M^{ImmEff}(id_R) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[a_1/f^M(1), ..., a_k/f^M(k)]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R^M_{data}(id_R, f^M(1), ..., f^M(k))$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$

(4) for each i :

$R^M_{exec}(id_R, x_1, ..., x_q) \wedge \pi_i(id_R) \rightsquigarrow R^M_{exec}(id_R, x_1, ..., x_q)[x_i/0]$

$R^M_{exec}(id_R, x_1, ..., x_q) \wedge \neg\pi_i(id_R) \rightsquigarrow R^M_{exec}(id_R, x_1, ..., x_q)[x_i/1]$

(5) [CopyRest]

}, where

(1) substitutes attributes' values with the payload of the message; (2) propagates values to the message hub; (3) blocks the artifact instance; (4) initializes the eligibility flags for each PAC-rule, where $\pi_i(id_R)$ is a prerequisite of the i-th PAC-rule.

Translation 2 (Immediate effect of 2-way service call generated by artifact instance).
Assume a 2-way service call type F within an atomic stage S_p, its associated artifact type R, input signature $(b_1 : Typ(b_1), ...b_l : Typ(b_l))$ and output signature $(a'_1 : Typ(a'_1), ...a'_k : Typ(a'_k))$ where $a'_i \in Att_{data}$.
Assume also a set of PAC-rules $\{(\pi_i, \alpha_i, \gamma_i)\}$.
Then a service call and an immediate effect of a service call return of type F on some artifact instance A of type R may be modeled by the following condition-action rule:

$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_p = true \wedge R_{block}(id_R, false) \mapsto$

$\alpha_F^{call}(id_R) :$

$\{$(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[a_1/f^F(\overline{b}, 1), ..., a_k/f^F(\overline{b}, k)]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R^F_{data}(id_R, f^F(\overline{b}, 1), ..., f^F(\overline{b}, k))$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$

(4) for each i :

$R^F_{exec}(id_R, x_1, ..., x_q) \wedge \pi_i(id_R) \rightsquigarrow R^F_{exec}(id_R, x_1, ..., x_q)[x_i/0]$

$R^F_{exec}(id_R, x_1, ..., x_q) \wedge \neg\pi_i(id_R) \rightsquigarrow R^F_{exec}(id_R, x_1, ..., x_q)[x_i/1]$

(5) [CopyRest]

}, where

(1) substitutes attributes' values with the result of the service call; (2) propagates values to the message hub; (3) blocks the artifact instance; (4) initializes the

eligibility flags for each PAC-rule, where $\pi_i(id_R)$ is a prerequisite of the i-th PAC-rule.

Translation 3 (PAC-1 rule (Activating a stage)).
Assume a stage S_j and its activating guard $g_j = [$ **on** $\xi(x)$ **if** $\phi(x)]$, where $\xi(x)$ is a triggering event and $\phi(x)$ is a condition. Then activating a stage S_j by validating g_j can be modeled by the following condition-action rule:
NB: Include term $(S' = true)$ if S' is parent of S_j.
NB: Note that prerequisite is checked on the stage of implementing immediate effect and, if not validated, will lead to marking x_k as 1, so will lead to skipping this CA-rule.
NB: Include effect propagating the outgoing message to the outgoing hub, if the stage to be activated is atomic and contains an action of sending a one-way message O with a signature $(b_1 : Typ(b_1), ...b_k : Typ(b_k))$, where $b_i \in Att_{data}$. .

$R_{exec}(id_R, \overline{x}) \wedge x_k = 0 \wedge exec(k) \wedge R_{block}(id_R, true) \mapsto$

$\alpha^k_{exec}(id_R, \overline{a'}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_j/true]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R^{S_j}_{chg}(id_R, true)$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R^{O}_{out}(id_R, b_1, ..., b_k)$

(4) $R_{exec}(id_R, \overline{x}) \wedge x_k = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_k/1]$

(5) [CopyMessagePools]

(6) [CopyRest] $\}$,

where $exec(k) = \bigwedge_k x_k$ such that $r_k <_{PDG} r_a$

$R_\xi(id_R, \overline{a'}) = R_M(id_R, \overline{a'})$ if the guard contains incoming message event or $R^{att}_{chg}(id_R, status_{new}))$ if the guard contains internal event.

(1) activates a stage on a condition; (2) propagates internal event of opening a stage on a condition; (3) prepares eventual outgoing message for sending; (4) flags the microstep as performed.

Translation 4 (PAC-2 rule (Milestone achiever)).
Assume a stage S_j and its milestone m_j with achieving sentry [**on** $\xi(x)$ **if** $\phi(x)]$, where $\xi(x)$ is a triggering event and $\phi(x)$ is a condition. Then achieving a mile-

stone m_j can be modeled by the following condition-action rule:

$R^M_{exec}(id_R, \overline{x}) \wedge x_k = 0 \wedge exec(k) \wedge R_{block}(id_R, true) \mapsto$

$\alpha^k_{exec}(id_R, \overline{a'}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[m_j/true]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R^{m_j}_{chg}(id_R, true)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_k = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_k/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$,

where $exec(k) = \bigwedge_k x_k$ such that $r_k <_{PDG} r_a$

$R_\xi(id_R, \overline{a'}) = R_M(id_R, \overline{a'})$ if the guard contains incoming message event

or $R^{att}_{chg}(id_R, status_{new})$) if the guard contains internal event.

(1) achieves a milestone on a condition; (2) propagates internal event of achieving a milestone on a condition; (3) flags the microstep as performed;

Translation 5 (PAC-3 rule (Milestone invalidator)).
Assume a stage S_j and its milestone m_j with invalidating sentry [**on** $\xi(x)$ **if** $\phi(x)$], where $\xi(x)$ is a triggering event and $\phi(x)$ is a condition. Then invalidating a milestone m_j can be modeled by the following condition-action rule:

$R^M_{exec}(id_R, \overline{x}) \wedge x_k = 0 \wedge exec(k) \wedge R_{block}(id_R, true) \mapsto$

$\alpha^k_{exec}(id_R, \overline{a'}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[m_j/false]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R^{m_j}_{chg}(id_R, false)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_k = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_k/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$,

where $exec(k) = \bigwedge_k x_k$ such that $r_k <_{PDG} r_a$

$R_\xi(id_R, \overline{a'}) = R_M(id_R, \overline{a'})$ if the guard contains incoming message event

or $R^{att}_{chg}(id_R, status_{new})$) if the guard contains internal event.

(1) invalidates a milestone on a condition; (2) propagates internal event of invalidating a milestone on a condition; (3) flags the microstep as performed;

Translation 6 (PAC-4 rule (Opening stage invalidating milestone)).
Assume a stage S_j and its milestone m_j. Then invalidating a milestone m_j caused

by opening a stage can be modeled by the following condition-action rule:

$$
\begin{aligned}
&R_{exec}(id_R, \overline{x}) \wedge x_k = 0 \wedge exec(k) \wedge R_{block}(id_R, true) \mapsto \\
&\quad \alpha^k_{exec}(id_R, \overline{a'}, \overline{x}) : \{ \\
&\quad (1)\ \ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S_j}_{chg}(id_R, true)) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[m_j/false] \\
&\quad (2)\ \ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S_j}_{chg}(id_R, true)) \rightsquigarrow R^{m_j}_{chg}(id_R, false) \\
&\quad (3)\ \ R^M_{exec}(id_R, \overline{x}) \wedge x_k = 0 \rightsquigarrow R^M_{exec}(id_R, \overline{x})[x_k/1] \\
&\quad (4)\ \ [\mathsf{CopyMessagePools}] \\
&\quad (5)\ \ [\mathsf{CopyRest}]\ \ \}, \\
&\quad \text{where } exec(k) = \bigwedge_k x_k \text{ such that } r_k <_{PDG} r_a
\end{aligned}
$$

(1) invalidates a milestone if the stage was open; (2) propagates internal event of invalidating a milestone; (4) flags the microstep as performed;

Translation 7 (PAC-5 rule (Closing a stage on achieving milestone)). Assume a stage S_j and its milestone m_j. Then closing a stage S_j caused by achieving a milestone m_j can be modeled by the following condition-action rule:

$$
\begin{aligned}
&R^M_{exec}(id_R, \overline{x}) \wedge x_k = 0 \wedge exec(k) \wedge R_{block}(id_R, true) \mapsto \\
&\quad \alpha^k_{exec}(id_R, \overline{a}, \overline{x}) : \{ \\
&\quad (1)\ \ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{m_j}_{chg}(id_R, true)) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_j/false] \\
&\quad (2)\ \ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{m_j}_{chg}(id_R, true)) \rightsquigarrow R^{S_j}_{chg}(id_R, false) \\
&\quad (3)\ \ R^M_{exec}(id_R, \overline{x}) \wedge x_k = 0 \rightsquigarrow R^M_{exec}(id_R, \overline{x})[x_k/1] \\
&\quad (4)\ \ [\mathsf{CopyMessagePools}] \\
&\quad (5)\ \ [\mathsf{CopyRest}]\ \ \}, \\
&\quad \text{where } exec(k) = \bigwedge_k x_k \text{ such that } r_k <_{PDG} r_a
\end{aligned}
$$

(1) closes a stage if the milestone was achieved; (2) propagates internal event of closing a stage; (4) flags the microstep as performed;

Translation 8 (PAC-6 rule (No activity in a closed stage)). Assume a stage S_j and its parent stage S'. Then closing a stage S_j caused by

closing its parent stage S' can be modeled by the following condition-action rule:

$$R^M_{exec}(id_R, \overline{x}) \wedge x_k = 0 \wedge exec(k) \wedge R_{block}(id_R, true) \mapsto$$
$$\alpha^k_{exec}(id_R, \overline{a'}, \overline{x}) : \{$$
(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S'}_{chg}(id_R, false)) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_j/false]$
(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S_j}_{chg}(id_R, false)) \rightsquigarrow R^{S_j}_{chg}(id_R, false)$
(3) $R^M_{exec}(id_R, \overline{x}) \wedge x_k = 0 \rightsquigarrow R^M_{exec}(id_R, \overline{x})[x_k/1]$
(4) [CopyMessagePools]
(5) [CopyRest] $\}$,
where $exec(k) = \bigwedge_k x_k$ such that $r_k <_{PDG} r_a$

(1) closes a stage if the parent stage is closed; (2) propagates internal event of closing a stage; (3) flags the microstep as performed;

Translation 9 (Sending outgoing messages to the environment and flushing the message hubs).
Assume a set of 1-way outgoing message types O_j obtained after all the PAC rules have been already taken into consideration. Then the conclusive part of a B-step, involving sending one-way outgoing messages and flushing the system message hubs may be modeled by the following CA-rule:

$$\exists R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \wedge R_{block}(id_R, true) \mapsto$$
$$\alpha^{send}_{flush}(id_R) : \{$$
(1) $R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \rightsquigarrow R_{block}(id_R, false)$
(2) $R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \rightsquigarrow R_{exec}(id_R, \overline{0})$
(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})$
(4) for each j :
$R^{O_j}_{out}(id_R, b_1, ..., b_k) \rightsquigarrow R_{result}(id_R, f^{O_j}(b_1, ..., b_k))$
(5) [CopyRest]

(1) unblocks the artifact instance; (2) flushes the eligibility flags; (3) copies data; (4) sends outgoing messages to the environment.

Translation 10 (Create artifact service call).
Assume a particular kind of a 2-way service call - create artifact service call, within an atomic stage S_p. Assume also a set of PAC-rules $\{(\pi_i, \alpha_i, \gamma_i)\}$.
Then a create artifact service call and its immediate effect of a service call return

may be modeled by the following condition-action rule:

$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_p = true \wedge R_{block}(id_R, false)$
$\mapsto \alpha_A^{create}(id_R, num)$:
{(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(f_A^{create}(\tilde{a}), \tilde{a}, \overline{0}, \overline{0})$
(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{data}^{srv-newA}(f_A^{create}(\tilde{a}))$
(4) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$
(5) for each i :
$R_{exec}^F(id_R, x_1, ..., x_q) \wedge \pi_i(id_R) \rightsquigarrow R_{exec}^F(id_R, x_1, ..., x_q)[x_i/0]$
$R_{exec}^F(id_R, x_1, ..., x_q) \wedge \neg\pi_i(id_R) \rightsquigarrow R_{exec}^F(id_R, x_1, ..., x_q)[x_i/1]$
(6) [CopyRest]
}, where $f_A^{create}(\tilde{a})$ returns ID of newly create artifact.

Translation 11 (Remove artifact service call).
Assume a particular kind of a 2-way service call - remove artifact service call, within an atomic stage S_p. Assume also a set of PAC-rules $\{(\pi_i, \alpha_i, \gamma_i)\}$.
Then a remove artifact service call and its immediate effect of a service call return may be modeled by the following condition-action rule:

$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_p = true \wedge R_{block}(id_R, false)$
$\mapsto \alpha_A^{remove}(id_R, num)$:
(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{data}^{srv-remA}(f_A^{remove}(\overline{a}))$
(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$
(5) for each i :
$R_{exec}^F(id_R, x_1, ..., x_q) \wedge \pi_i(id_R) \rightsquigarrow R_{exec}^F(id_R, x_1, ..., x_q)[x_i/0]$
$R_{exec}^F(id_R, x_1, ..., x_q) \wedge \neg\pi_i(id_R) \rightsquigarrow R_{exec}^F(id_R, x_1, ..., x_q)[x_i/1]$
(6) [CopyRest]
}, where $f_A^{remove}(\overline{a})$ returns the outcome of deletion.

4 Trivial example

Let's consider a process $Func$ which is as simple as calculating a sum $a + b$, given that $a \neq b$. The GSM concrete model of such process is, in fact, represented by the first stage on the Figure 2.

Then the corresponding artifact type has the following form:

$$(R, x, Att_{data} \cup Att_{status}, Typ, Stg, Mst, Lcyc), \text{ where}$$
$$Att_{data} = \{A_{ID}, a, b, c\}^3$$
$$Att_{status} = \{s_1, m_1, m_2\}^4, \text{ all Boolean}$$
$$Type = \{(a, Float), (b, Float), (c, Float)\}$$
$$S = \{s_1\}, \quad M = \{m_1, m_2\}$$

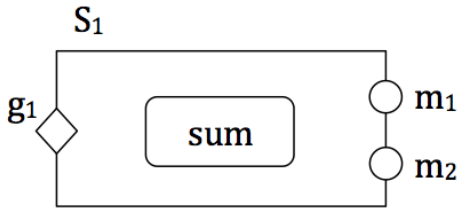

Fig. 2. GSM model of $(a + b)$

Consequently, lifecycle $Lcyc$ has the following form:

$$
\begin{aligned}
&(Substages, Task, Owns, Guards, Ach, Inv), \text{ where}\\
&\quad Substages = \emptyset\\
&\quad Task = \{(s_1, Sum)\}\\
&\quad Owns = \{(s_1, \{m_1, m_2\})\}\\
&\quad Guards = \{(s_1, \{\widetilde{g}_1\})\}\\
&\quad Ach = \{(m_1, \{\widetilde{m}_1\}), (m_2, \{\widetilde{m}_2\})\}\\
&\quad Inv = \emptyset
\end{aligned}
$$

Corresponding sentries for guards and milestones are given below:

$$
\begin{aligned}
\widetilde{g}_1 &: \text{on } x.Func^{call}(a, b) \text{ if } a \neq b\\
\widetilde{m}_1 &: \text{on } x.Sum^{return}(c) \text{ if } c \geq 0\\
\widetilde{m}_2 &: \text{if } c < 0
\end{aligned}
$$

Intuitively, the workflow of the given process is the following:

- after receiving a request from the environment, if the operands are not equal, the first stage is activated.
- the task associated with the first atomic stage is executed, calling the external service Sum with given parameters and obtaining the result;
- upon completing the request, if the result is positive, then milestone m_1 is achieved.
- if attribute storing the result of operation is negative, then milestone m_2 is achieved. Note that the corresponding sentry $\widetilde{m}_2$ doesn't contain event expression of receiving service call return. Thus, if the stage gets activated with $c < 0$, this milestone will be achieved immediately.

[1] For the sake of simplicity, we will omit such attributes as $mostRecEventType$ and $mostRecEventTime$

[2] Similarly, we will omit attributes like $m^{mostRecentUpdate}$ and $active_S^{mostRecentUpdate}$

PAC rules

Type	ID	Prerequisite	Antecedent	Consequent
PAC-1	x1	$\neg x.s_1$	**on** $x.Func^{call}(a,b)$ **if** $a \neq b$	$+x.s_1$
PAC-2	x2	$x.s_1$	**on** $x.Sum^{return}(c)$ **if** $c \geq 0$	$+x.m_1$
PAC-2	x3	$x.s_1$	**if** $c < 0$	$+x.m_2$
PAC-4	x4	$x.m_1$	**on** $+x.s_1$	$-x.m_1$
PAC-4	x5	$x.m_2$	**on** $+x.s_1$	$-x.m_2$
PAC-5	x6	$x.s_1$	**on** $+x.m_1$	$-x.s_1$
PAC-5	x7	$x.s_1$	**on** $+x.m_2$	$-x.s_1$

DCDS Translation

Data layer

The corresponding data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ will have the following form:

- $\mathcal{C} = \bigcup_i \mathbf{DOM}(Typ(Att_i))$,
- $\mathcal{R} = \{R_{att}\} \cup \{R^{m_i}_{chg} | m_i \in Mst\} \cup \{R^{S_j}_{chg} | S_j \in Stg\} \cup$
  $\cup\ \{R^{msg_k}_{data} | msg_k \in \mathbf{MSG}$ and msg_k is incoming 1-way message$\} \cup$
  $\cup\ \{R^{srv_p}_{data} | srv_p \in \mathbf{SRV}$ and srv_p is a service call return$\} \cup$
  $\cup\ \{R^{msg_q}_{out} | msg_q \in \mathbf{MSG}$ and msg_q is outgoing 1-way message$\} \cup$
  $\cup\ \{R_{exec}, R_{block}\}$,

  where R_{att} stores the attributes of an artifact, R_{exec} keeps information on which PAC rules have been taken in consideration while other relations are used to model the incoming / outgoing message pool:
  - $R_{att} = (id_A, a, b, c, s_1, m_1, m_2)$.
  - $R_{exec} = (id_A, x1, x2, x3, x4, x5, x6, x7)$.
  - $R_{block} = (id_A, blocked)$.
  - $R^{s_1}_{chg} = (id_A, newstate)$.
  - $R^{m_1}_{chg} = (id_A, newstate)$.
  - $R^{m_2}_{chg} = (id_A, newstate)$.
  - $R^{Func}_{data} = (id_A, a, b)$.
  - $R^{Sum}_{data} = (id_A, c)$.
- $\mathcal{E} = \emptyset$.
- $\mathcal{I}_0 = \emptyset$.

Immediate effect rules

Incoming message $Func$:

$$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto$$

$$\alpha^{ImmEff}_{Func}(id_R) : \{$$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[a/f^{Func}(1), b/f^{Func}(2)]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R^{Func}_{data}(id_R, f^{Func}(1), f^{Func}(2))$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$

(4)

$R_{exec}(id_R, \overline{x}) \wedge \neg S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/0, x2/1, x3/1, x6/1, x7/1]$

$R_{exec}(id_R, \overline{x}) \wedge S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/1, x2/0, x3/0, x6/0, x7/0]$

$R_{exec}(id_R, \overline{x}) \wedge m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/0]$

$R_{exec}(id_R, \overline{x}) \wedge \neg m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/1]$

$R_{exec}(id_R, \overline{x}) \wedge m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/0]$

$R_{exec}(id_R, \overline{x}) \wedge \neg m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/1]$

(5) [CopyRest]

}

Service call return Sum:

$$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_1 = true \wedge R_{block}(id_R, false) \mapsto$$

$$\alpha^{call}_{Sum}(id_R) : \{$$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[c/f^{Sum}(a, b, 1)]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R^{Sum}_{data}(id_R, f^{Sum}(a, b, 1))$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$

(4)

$R_{exec}(id_R, \overline{x}) \wedge \neg S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/0, x2/1, x3/1, x6/1, x7/1]$

$R_{exec}(id_R, \overline{x}) \wedge S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/1, x2/0, x3/0, x6/0, x7/0]$

$R_{exec}(id_R, \overline{x}) \wedge m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/0]$

$R_{exec}(id_R, \overline{x}) \wedge \neg m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/1]$

$R_{exec}(id_R, \overline{x}) \wedge m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/0]$

$R_{exec}(id_R, \overline{x}) \wedge \neg m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/1]$

(5) [CopyRest]

}

PAC rules

PAC-1 rule x_1:

$R_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^1_{exec}(id_R, a, b, \overline{x}) : \{$

(1) $R^{Func}_{data}(id_R, a, b) \wedge R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge a \neq b \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_1/true]$

(2) $R^{Func}_{data}(id_R, a, b) \wedge R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge a \neq b \rightsquigarrow R^{S_1}_{chg}(id_R, true)$

(3) $R^{M}_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \rightsquigarrow R^{M}_{exec}(id_R, \overline{x})[x_1/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$

PAC-2 rule x_2:

$R_{exec}(id_R, \overline{x}) \wedge x_2 = 0 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^2_{exec}(id_R, c, \overline{x}) : \{$

(1) $R^{Sum}_{data}(id_R, c) \wedge R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge c \geq 0 \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[m_1/true]$

(2) $R^{Sum}_{data}(id_R, c) \wedge R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge c \geq 0 \rightsquigarrow R^{m_1}_{chg}(id_R, true)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_2 = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_2/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$

PAC-2 rule x_3:

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_3 = 0 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^3_{exec}(id_R, \overline{a}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge c < 0 \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[m_2/true]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge c < 0 \rightsquigarrow R^{m_2}_{chg}(id_R, true)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_3 = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_3/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$

PAC-4 rule x_4:

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_4 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^4_{exec}(id_R, \overline{a}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S_1}_{chg}(id_R, true)) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[m_1/false]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S_1}_{chg}(id_R, true)) \rightsquigarrow R^{m_1}_{chg}(id_R, false)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_4 = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_4/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$

PAC-4 rule x_5:

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_5 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^5_{exec}(id_R, \overline{a}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S_1}_{chg}(id_R, true)) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[m_2/false]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{S_1}_{chg}(id_R, true)) \rightsquigarrow R^{m_2}_{chg}(id_R, false)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_5 = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_5/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$

PAC-5 rule x_6:

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_6 = 0 \wedge x_2 = 1 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^6_{exec}(id_R, \overline{a}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{m_1}_{chg}(id_R, true)) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_1/false]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{m_1}_{chg}(id_R, true)) \rightsquigarrow R^{S_1}_{chg}(id_R, false)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_6 = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_6/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$

PAC-5 rule x_7:

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_7 = 0 \wedge x_3 = 1 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^7_{exec}(id_R, \overline{a}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{m_2}_{chg}(id_R, true)) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_1/false]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R^{m_2}_{chg}(id_R, true)) \rightsquigarrow R^{S_1}_{chg}(id_R, false)$

(3) $R_{exec}(id_R, \overline{x}) \wedge x_7 = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_7/1]$

(4) [CopyMessagePools]

(5) [CopyRest] $\}$

Sending outgoing messages and unblocking the artifact instance

$\exists R_{exec}(id_R, \overline{x}) \wedge \forall i \; x_i = 1 \wedge R_{block}(id_R, true) \mapsto$

$\alpha^{send}_{flush}(id_R) : \{$

(1) $R_{exec}(id_R, \overline{x}) \wedge \forall i \; x_i = 1 \rightsquigarrow R_{block}(id_R, false)$

(2) $R_{exec}(id_R, \overline{x}) \wedge \forall i \; x_i = 1 \rightsquigarrow R_{exec}(id_R, \overline{0})$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})$

(4) for each j :

$R^{O_j}_{out}(id_R, b_1, ..., b_k) \rightsquigarrow R_{result}(id_R, f^{O_j}(b_1, ..., b_k))$

(5) [CopyRest]

4.1 Constructing a transition system (TS) from DCDS encoding

Let us now try to simulate the construction of a transition system resulting from the obtained translation. We start with an initial state I_0 such that:

- $s_1 = m_1 = m_2 = 0$
- $R_{block}(id_R, false)$
- $R_{exec}(id_R, \overline{0})$
- $R^{Sum}_{data} = R^{Func}_{data} = \emptyset$

Let us evaluate the condition part of all CA-rules and mark with $(*)$ applicable ones:

$(*)$ $\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto \{\}$
$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_1 = true \wedge R_{block}(id_R, false) \mapsto \{\}$
$R_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$
$R_{exec}(id_R, \overline{x}) \wedge x_2 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$
$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_3 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$
$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_4 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$
$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_5 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$
$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_6 = 0 \wedge x_2 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$
$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_7 = 0 \wedge x_3 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$
$\exists R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$

So, only the first rule is applicable, let us check what's inside it:

$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto$
$\alpha^{ImmEff}_{Func}(id_R) : \{$
(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[a/f^{Func}(1), b/f^{Func}(2)]$
(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R^{Func}_{data}(id_R, f^{Func}(1), f^{Func}(2))$
(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$
(4)
$R_{exec}(id_R, \overline{x}) \wedge \neg S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/0, x2/1, x3/1, x6/1, x7/1]$
$R_{exec}(id_R, \overline{x}) \wedge S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/1, x2/0, x3/0, x6/0, x7/0]$

$R_{exec}(id_R, \overline{x}) \wedge m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/0]$
$R_{exec}(id_R, \overline{x}) \wedge \neg m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/1]$

$R_{exec}(id_R, \overline{x}) \wedge m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/0]$
$R_{exec}(id_R, \overline{x}) \wedge \neg m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/1]$
(5) [CopyRest]
$\}$

First effect is to obtain input from the environment by calling the function f^{Func}. Second effect propagates obtained data to the inner message pool. (3) Blocks the artifact instance. Effects from (4) decide which CA-rules are relevant. For our current state, we get that: $x1 = 0$ and all the rest $x_i = 1$. Which means that only $x1$ is relevant.

Thus, we get the following situation:

- $s_1 = m_1 = m_2 = 0$
- $R_{block}(id_R, true)$
- $R_{exec}(id_R, (0, 1, ..., 1))$
- $R^{Sum}_{data} = \emptyset, R^{Func}_{data} = (a, b)$

Let us evaluate again the condition part of all CA-rules and mark with $(*)$ applicable ones:

$$
\begin{array}{ll}
 & \exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto \{\} \\
 & \exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_1 = true \wedge R_{block}(id_R, false) \mapsto \{\} \\
(*) & R_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \wedge R_{block}(id_R, true) \mapsto \{\} \\
 & R_{exec}(id_R, \overline{x}) \wedge x_2 = 0 \wedge R_{block}(id_R, true) \mapsto \{\} \\
 & R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_3 = 0 \wedge R_{block}(id_R, true) \mapsto \{\} \\
 & R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_4 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\} \\
 & R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_5 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\} \\
 & R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_6 = 0 \wedge x_2 = 1 \wedge R_{block}(id_R, true) \mapsto \{\} \\
 & R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_7 = 0 \wedge x_3 = 1 \wedge R_{block}(id_R, true) \mapsto \{\} \\
 & \exists R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \wedge R_{block}(id_R, true) \mapsto \{\}
\end{array}
$$

So, only the third rule is applicable, let us check what's inside it:

$$
\begin{array}{l}
R_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \wedge R_{block}(id_R, true) \mapsto \\
\quad \alpha^1_{exec}(id_R, a, b, \overline{x}) : \{ \\
\quad (1)\ \ R^{Func}_{data}(id_R, a, b) \wedge R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge a \neq b \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_1/true] \\
\quad (2)\ \ R^{Func}_{data}(id_R, a, b) \wedge R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge a \neq b \rightsquigarrow R^{S_1}_{chg}(id_R, true) \\
\quad (3)\ \ R^{M}_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \rightsquigarrow R^{M}_{exec}(id_R, \overline{x})[x_1/1] \\
\quad (4)\ \ [\mathsf{CopyMessagePools}] \\
\quad (5)\ \ [\mathsf{CopyRest}]\ \ \}
\end{array}
$$

At this point there are 2 possible cases – when $a \neq b$ and $a = b$. If $a = b$ the first 2 effects are ignored and we just mark $x_1 = 1$ and that's it. Then we go to the last rule which unblocks the artifact instance.
If $a \neq b$, then we open a stage s_1 and propagate this event to the inner message pool. We also mark $x_1 = 1$. So, we have the following situation:

- $s_1 = 1, m_1 = m_2 = 0$

- $R_{block}(id_R, true)$
- $R_{exec}(id_R, (1, 1, ..., 1))$
- $R^{Sum}_{data} = \emptyset, R^{Func}_{data} = (a, b)$

Let us evaluate again the rules:

$$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto \{\}$$
$$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_1 = true \wedge R_{block}(id_R, false) \mapsto \{\}$$
$$R_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$$
$$R_{exec}(id_R, \overline{x}) \wedge x_2 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$$
$$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_3 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$$
$$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_4 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$$
$$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_5 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$$
$$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_6 = 0 \wedge x_2 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$$
$$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_7 = 0 \wedge x_3 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$$
$$(*)\quad \exists R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$$

The last rule is applicable:

$$\exists R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \wedge R_{block}(id_R, true) \mapsto$$
$$\alpha^{send}_{flush}(id_R) : \{$$
$$(1)\quad R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \rightsquigarrow R_{block}(id_R, false)$$
$$(2)\quad R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \rightsquigarrow R_{exec}(id_R, \overline{0})$$
$$(3)\quad R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})$$
$$(4)\quad \text{for each } j :$$
$$R^{O_j}_{out}(id_R, b_1, ..., b_k) \rightsquigarrow R_{result}(id_R, f^{O_j}(b_1, ..., b_k))$$
$$(5)\quad [\text{CopyRest}]$$

So we have:

- $s_1 = 1, m_1 = m_2 = 0$
- $R_{block}(id_R, false)$
- $R_{exec}(id_R, (0, 0, ..., 0))$
- $R^{Sum}_{data} = \emptyset, R^{Func}_{data} = \emptyset$

Let us evaluate again the rules:

$(*)$ $\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto \{\}$

$(*)$ $\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge S_1 = true \wedge R_{block}(id_R, false) \mapsto \{\}$

$R_{exec}(id_R, \overline{x}) \wedge x_1 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$

$R_{exec}(id_R, \overline{x}) \wedge x_2 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_3 = 0 \wedge R_{block}(id_R, true) \mapsto \{\}$

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_4 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_5 = 0 \wedge x_1 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_6 = 0 \wedge x_2 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$

$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{exec}(id_R, \overline{x}) \wedge x_7 = 0 \wedge x_3 = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$

$\exists R_{exec}(id_R, \overline{x}) \wedge \forall i\ x_i = 1 \wedge R_{block}(id_R, true) \mapsto \{\}$

Here comes indeterministic choice of the rule to apply. Let us, say, apply the first one:

$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto$

$\alpha_{Func}^{ImmEff}(id_R) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[a/f^{Func}(1), b/f^{Func}(2)]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{data}^{Func}(id_R, f^{Func}(1), f^{Func}(2))$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$

(4)

$R_{exec}(id_R, \overline{x}) \wedge \neg S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/0, x2/1, x3/1, x6/1, x7/1]$

$R_{exec}(id_R, \overline{x}) \wedge S_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x1/1, x2/0, x3/0, x6/0, x7/0]$

$R_{exec}(id_R, \overline{x}) \wedge m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/0]$

$R_{exec}(id_R, \overline{x}) \wedge \neg m_1 \rightsquigarrow R_{exec}(id_R, \overline{x})[x4/1]$

$R_{exec}(id_R, \overline{x}) \wedge m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/0]$

$R_{exec}(id_R, \overline{x}) \wedge \neg m_2 \rightsquigarrow R_{exec}(id_R, \overline{x})[x5/1]$

(5) [CopyRest]

$\}$

Then what we get is:

- $s_1 = 1, m_1 = m_2 = 0$
- $R_{block}(id_R, true)$
- $R_{exec}(id_R, (1, 0, 0, 1, 1, 0, 0))$
- $R_{data}^{Sum} = \emptyset, R_{data}^{Func} = (a, b)$

So we have to apply $x2, x3, x6, x7$.

5 Some important proofs: soundness and completeness

The proof plan:

1. Prove that for each micro-step of the GSM model, the corresponding DCDS CA-rule results in the same state (pre-snapshot) of the model, w.r.t. data and status attributes.
2. Prove that for each GSM B-step (certain path in the resulting transition system) there exists a corresponding execution in the DCDS transition system.
3. Prove that for each execution path in the DCDS transition system, there exists a corresponding one in GSM.

Lemma 1. *For each micro-step, consisting of applying a ground PAC rule* $(\pi_k, \alpha_k, \gamma_k)$ *to a pre-snapshot* Σ_j*, the corresponding translation of this rule – a DCDS condition-action rule* $Q_k \mapsto \alpha_k(p_1, ..., p_q) : \{e_1, ..., e_m\}$*, results in the same pre-snapshot* Σ_{j+1} *w.r.t. data and status attributes.*

Proof. First, we have to prove that the CA-rule, corresponding to computing the $\Sigma_1 = ImmEffect(\Sigma, e, t)$, results in the same pre-snapshot as $ImmEffect(\Sigma, e, t)$. By definition of the immediate effect of an incoming event $e = M(A_1 : c_1, ..., A_n : c_n)$, $ImmEffect(\Sigma, e, t)$ is a pre-snapshot Σ' obtained from Σ by modifying the corresponding artifact instance I in the following way [5]: for each data attribyte A_i, set $I.A_i := c_i$.

Let us now consider the corresponding CA-rule:

$$\exists \overline{a}, \overline{s}, \overline{m}\ R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_{block}(id_R, false) \mapsto$$
$$\alpha_M^{ImmEff}(id_R) : \{$$
(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[a_1/f^M(1), ..., a_k/f^M(k)]$
(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R^M_{data}(id_R, f^M(1), ..., f^M(k))$
(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow R_{block}(id_R, true)$
(4) for each i :
$R^M_{exec}(id_R, x_1, ..., x_q) \wedge \pi_i(id_R) \rightsquigarrow R^M_{exec}(id_R, x_1, ..., x_q)[x_i/0]$
$R^M_{exec}(id_R, x_1, ..., x_q) \wedge \neg\pi_i(id_R) \rightsquigarrow R^M_{exec}(id_R, x_1, ..., x_q)[x_i/1]$
(5) [CopyRest]
}, where

The condition part of this CA-rule selects an artifact instance and checks whether it is able to process the message, i.e. is not busy with processing another one. Then, the first effect of the action (1) does exactly what is required by the definition of $ImmEffect$ – changes the data attributes affected by the payload of e. The second effect (2) propagates the values of the incoming event to the system message hub, which is used by the DCDS engine to encode the execution of a model. Thus, we can abstract from it, as well from the third effect

[5] As mentioned earlier, we omit *mostRecEventType* and *mostRecEventTime*

(3), which blocks the artifact instance from receiving other messages until the current message is fully processed. Also the forth effect (4) may be abstracted away, since it is also a system information.
It should be noted, though, that the fourth effect (4) implements the step of selecting applicable CA-rules according to the incremental semantics of GSM. For those CA-rules whose prerequisite is valid ($\pi_i(id_R) == true$), the corresponding CA-rule is marked as 0, i.e. to be taken into consideration. Not eligible rules are marked with 1, i.e. already taken into consideration. Therefore, since:

- it is assumed that in GSM incoming messages "are processed by the artifact instances one at a time";
- corresponding CA-rule uses its own blocking mechanism to ensure this;
- the only effect changing data attributed is the first one (1), which is applied whenever an action is fire (i.e. without any condition);
- the first effect strictly corresponds to the definition of the $ImmEffect$ in GSM;
- no other effect involves neither data nor status attributes,

then it may be claimed that the DCDS pre-snapshot obtained after firing the corresponding CA-rule coincides with the GSM pre-snapshot $\Sigma_1 = ImmEffect(\Sigma, e, t)$ w.r.t. to data and status attributes. Similarly it can be shown for the case of service call return. The only difference would be a condition of an atomic stage to be activated in order to enable service call.

Now let us get down to proving correspondence between PAC rules and their translations.

Consider, for instance, a PAC-1 rule $(\pi_k, \alpha_k, \gamma_k)$ corresponding to a certain micro-step in the incremental foundation. We have to prove that a corresponding DCDS CA-rule is eligible to apply the effect γ if and only if the PAC-1 rule is eligible to apply the effect γ and that the effect of firing this rule will result in the coinciding pre-snapshot w.r.t. to data and status attributes.

The PAC-1 rule is eligible to apply the effect γ if and only if each of the following holds:

- $\Sigma \vDash \pi_k$, i.e. the prerequisite is met.
- $\Sigma_j \vDash \alpha_k$, for $\alpha_k = [$ **on** $\xi(x)$ **if** $\phi(x)]$, i.e. the antecedent is satisfied.
- the ordering implied by $PDG(\Gamma)$ is respected, i.e. for each pair (r, r') of ground rules with abstract actions $\odot R.s$ and $\odot' R'.s'$, respectively, if $\odot R.s < \odot' R'.s'$, then the rule r must be considered for firing before the rule r' is considered for firing.

Now let us consider the translation of the PAC-1 rule to DCDS CA-rule and show that the conditions for applying the effect γ coincide with those listed above. The corresponding CA-rule looks like the following:

$R_{exec}(id_R, \overline{x}) \wedge x_k = 0 \wedge exec(k) \wedge R_{block}(id_R, true) \mapsto$

$\alpha^k_{exec}(id_R, \overline{a'}, \overline{x}) : \{$

(1) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R_{att}(id_R, \overline{a}, \overline{s}, \overline{m})[S_j/true]$

(2) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R^{S_j}_{chg}(id_R, true)$

(3) $R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \wedge R_\xi(id_R, \overline{a'}) \wedge S' = true \wedge \phi(id_R) \rightsquigarrow R^{O}_{out}(id_R, b_1, ..., b_k)$

(4) $R_{exec}(id_R, \overline{x}) \wedge x_k = 0 \rightsquigarrow R_{exec}(id_R, \overline{x})[x_k/1]$

(5) [CopyMessagePools]

(6) [CopyRest] $\}$,

where $exec(k) = \bigwedge_k x_k$ such that $r_k <_{PDG} r_a$

$R_\xi(id_R, \overline{a'}) = R_M(id_R, \overline{a'})$ if the guard contains incoming message event or $R^{att}_{chg}(id_R, status_{new})$) if the guard contains internal event.

The condition part of this CA-rule obtains the current state of the execution plan for this event and insures that this CA-rule has not yet been taken into consideration ($x_k = 0$) and that all the preceding CA-rules have been taking into consideration ($exec(k)$).

Assume that $\Sigma \nvDash \pi_k$. Then, since each micro-step is preceded by the $ImmEffect$, then the CA-rule implementing the immediate effect of the event has already been fired. Since it has been fired then for each i either of the effects has been applied:

$$R^F_{exec}(id_R, x_1, ..., x_q) \wedge \pi_i(id_R) \rightsquigarrow R^F_{exec}(id_R, x_1, ..., x_q)[x_i/0]$$
$$R^F_{exec}(id_R, x_1, ..., x_q) \wedge \neg\pi_i(id_R) \rightsquigarrow R^F_{exec}(id_R, x_1, ..., x_q)[x_i/1]$$

Since $\Sigma \nvDash \pi_k$, then it is the second effect that has been applied, so $x_k = 1$, which prevents our CA-rule to fire, since the condition part is not met.

Now let us assume that the prerequisite holds, but the ordering implied by $PDG(\Gamma)$ is not yet respected, i.e. there exists a rule r', such that $\odot' R'.s' < +R.active_S$ and that it has not yet been considered. Then the executability flag x' will be equal to 0, which will prevent rule from firing, since the condition part of the CA-rule contains a check $x' = 1$.

Now let us assume that $\Sigma \vDash \pi_k$, the ordering is respected, but $\Sigma_j \nvDash \alpha_k$, so either **on** $\xi(x)$ hasn't happened or $\Sigma \nvDash \phi_k$. Then the CA-rule will be eligible to fire, however, the effects (1-3) will not be applied, since the query for instantiating them will be empty:

- in case $\Sigma_j \nvDash \phi_k$ – it is obvious.
- in case **on** $\xi(x)$ hasn't happened, this means that the corresponding record hasn't been put into the R_ξ, therefore R_ξ will be empty.

. Then the only effect that will possible take place are (3) - (6), which do not deal with any data or status attributes and, however, mark this PAC-rule as considered, so that rules dependent on this one could proceed.

Not let us assume that $\Sigma \vDash \pi_k$, $\Sigma_j \vDash \alpha_k$ and the ordering is respected. Then the effect will be applied and will result in toggling the status attribute $R.active_S$ to *true*. None of the remaining effects deal with data or status attribute, so the resulting DCDS pre-snapshot will coincide to that of GSM micro-step w.r.t. to data and status attributes.

The proof for other PAC rules can be formulated similarly to PAC-1.

The proof for CA-rule of sending a set of outgoing one-way messages once all the PAC rules have been taken into consideration, can be formulated similarly to *ImmEffect* rule.

■

We now have to prove the second and the third statement of the proof plan.

Lemma 2. *Given an artifact instance A_R for each possible GSM B-step (i.e. a sequence of micro-steps preceded by ImmEff and followed by the step of sending outgoing messages to the environment) there exists a corresponding execution in the DCDS transition system.*

Proof. In order to prove this statement we have to prove that the mechanism used by the DCDS translation to restrict the possible sequences of CA-rules results in the same order imposed by the PDG graph of the GSM model.

Let us assume we have a sequence of PAC-rules $\Gamma_{PAC} = \{r_i = (\pi_i, \alpha_i, \gamma_i)\}$, preceded by the *ImmEffect* micro-step, which respect the order imposed by the PDG graph constructed for the given GSM model. We now have to prove that for the set of corresponding CA-rules $\{tr(r_i)\}$ the following holds:

$\forall r_m, r_n \in \Gamma_{PAC}$ if $r_m <_{PDG} r_n$ then for any path in DCDS transition system, $tr(r_m)$ is considered for firing before $tr(r_n)$ is considered for firing.

Assume $r_n = (\pi, \alpha, \odot R.s)$ and $r_m = (\pi', \alpha', \odot' R'.s')$. Then $\odot R.s < \odot' R'.s'$. This means that, by construction of PDG, α' contains $\odot R.s$ as a triggering event (or contains $R.s$ in its condition).

Let us now consider corresponding DCDS CA-rules:

$$tr(r_m) = Q \mapsto act$$
$$tr(r_n) = Q' \mapsto act'$$

By definition of DCDS translation, $exec(n) \in Q'$ where:

for each PAC rule $r_k = (\pi_k, \alpha_k, \gamma_k)$ the expression $exec(k)$ for the corresponding CA-rule $tr(r_k)$ is defined as follows:

$$exec(k) = \bigwedge_j x_j \text{ such that } r_j <_{PDG} r_k \text{ (i.e. } \gamma_j \in \alpha_k),$$

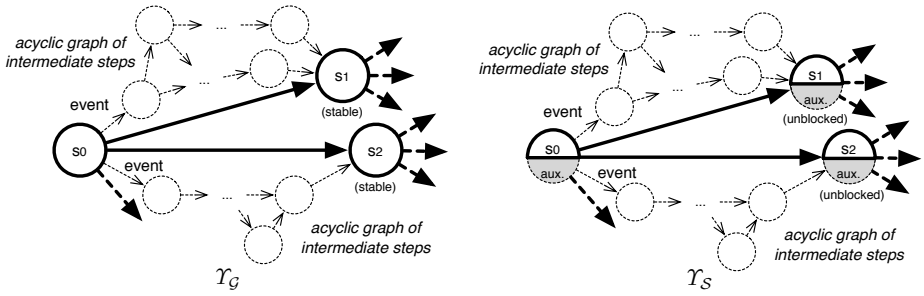

Fig. 3. Construction of the B-step transition system $\Upsilon_{\mathcal{G}}$ and unblocked-state transition system $\Upsilon_{\mathcal{S}}$ for a GSM model $\mathcal{G}$ with initial snapshot s_0 and the corresponding DCDS $\mathcal{S}$

where x_j is a boolean flag that shows whether the CA-rule $tr(r_j)$ has been taken into consideration. By construction of the DCDS translation, the boolean flag x_j is only affected in the CA-rule $tr(r_j)$ and in the first micro-step incorporating the immediate effect. The $ImmEff$ micro-step checks the prerequisite of a PAC rule in order to set the value of x_j. Assume it is set to 1, which means that prerequisite is not satisfied and therefore r_j cannot influence α_k, so r_k can fire, which is totally valid. Assume now it is set to 0, which means that r_j is applicable and should be taken into consideration. Then, the only place in the translation it may be affected is the $tr(r_j)$ and it will be, in fact, changed to 1, whenever this CA-rule will be nondeterministically chosen to fire. Till then it will be 0, which will prevent $tr(r_k)$ from firing.

Therefore $tr(r_n)$ will not be taken into consideration unless $tr(r_m)$ has been taken into consideration.

■

Lemma 3. *Given an artifact instance A_R, its GSM model and a corresponding DCDS translation, for each possible execution in DCDS starting with Immediate Effect rule, there exists a corresponding B-step in GSM model, which results in the same next pre-snapshot Σ_{j+1} w.r.t. data and status attributes.*

Proof. The proof is done by construction of CA-rules, see Section 3.2.

Given a GSM model $\mathcal{G}$ with initial snapshot S_0, we denote by $\Upsilon_{\mathcal{G}}$ its *B-step transition system*, i.e., the infinite-state transition system obtained by iteratively applying the incremental GSM semantics starting from S_0 and nondeterministically considering each possible incoming event. The states of $\Upsilon_{\mathcal{G}}$ correspond to stable snapshots of $\mathcal{G}$, and each transition corresponds to a B-step. We abstract away from the single micro-steps constituting a B-step, because they represent temporary intermediate states that are not interesting for verification purposes. Similarly, given the DCDS $\mathcal{S}$ obtained from the translation of $\mathcal{G}$, we denote by $\Upsilon_{\mathcal{S}}$ its *unblocked-state transition system*, obtained by starting from S_0, and iteratively applying nondeterministically the CA-rules of the process, and the

corresponding actions, in all the possible ways. As for states, we only consider those database instances where all artifact instances are not blocked; these correspond in fact to stable snapshots of $\mathcal{G}$. We then connect two such states provided that there is a sequence of (intermediate) states that lead from the first to the second one, and for which at least one artifact instance is blocked; these sequence corresponds in fact to a series of intermediate-steps evolving the system from a stable state to another stable state. Finally, we project away all the auxiliary relations introduced by the translation mechanism, obtaining a *filtered* version of $\Upsilon_{\mathcal{S}}$, which we denote as $\Upsilon_{\mathcal{S}}|_{\mathcal{G}}$. The intuition about the construction of these two transition systems is given in Figure 3. Notice that the intermediate micro-steps in the two transition systems can be safely abstracted away because: (i) thanks to the toggle-once principle, they do not contain any "internal" cycle; (ii) respecting the firing order imposed by $\mathcal{G}$, they all lead to reach the same next stable/unblocked state.

We can then establish the one-to-one correspondence between these two transition systems by applying subsequently results obtained from Lemmas 1 – 3 to prove the following theorem:

Theorem 1 (Soundness and completeness).
Given a GSM model $\mathcal{G}$ and its translation into a corresponding DCDS $\mathcal{S}$, the corresponding B-step transition system $\Upsilon_{\mathcal{G}}$ and filtered unblocked-state transition system $\Upsilon_{\mathcal{S}}|_{\mathcal{G}}$ are equivalent, i.e., $\Upsilon_{\mathcal{G}} \equiv \Upsilon_{\mathcal{S}}|_{\mathcal{G}}$.

6 State-bounded GSM models

The sound and complete translation of a GSM model into a corresponding DCDS provides the basis for applying the decidability and complexity results discussed in [9] for a purely technical DCDS framework, to more business-intuitive GSM approach. In particular, we make use of a semantic condition posed on the infinite-state transition system representing the execution semantics of the DCDS under study – *state-boundedness* – that guarantees decidability for a rich variant of first-order μ-calculus with some limitation on quantification over time [9].

We now take advantage of the key decidability result obtained for DCDSs and study verifiability of *state-bounded GSM models*. Observe that state-boundedness is not a too restrictive condition. It requires each state of the transition system to contain a bounded number of tuples. However, this does not mean that the system in general is restricted to encounter only a limited amount of data: infinitely many values may be distributed *across* the states (i.e. along an execution), provided that they do not accumulate in the same state. Furthermore, infinitely many executions are supported, reflecting that whenever an external event updates a slot of the information system maintained by a GSM artifact, infinitely many successor states in principle exist, each one corresponding to a specific new value for that slot.

Lemma 4. Given a GSM model $\mathcal{G}$ and its DCDS translation $\mathcal{S}$, $\mathcal{G}$ is state-bounded if and only if $\mathcal{S}$ is state-bounded.

Proof. Recall that $\mathcal{S}$ contains some auxiliary relations, used to restrict the applicability of CA-rules in order to enforce the execution assumptions of GSM:

- the eligibility tracking table R_{exec},
- the artifact instance blocking flags R_{block},
- the internal message pools $R_{data}^{msg_k}$, $R_{data}^{srv_p}$, $R_{out}^{msg_q}$, and
- the tables of status changes $R_{chg}^{m_i}$, $R_{chg}^{s_j}$.

We discuss the two implications separately. ($\Leftarrow$) This is directly obtained by observing that, if $\Upsilon_{\mathcal{S}}$ is state-bounded, then also $\Upsilon_{\mathcal{S}}|_{\mathcal{G}}$ is state-bounded. From Theorem 1, we know that $\Upsilon_{\mathcal{S}}|_{\mathcal{G}} \equiv \Upsilon_{\mathcal{G}}$, and therefore $\Upsilon_{\mathcal{G}}$ is state-bounded as well. ($\Rightarrow$) We have to show that state boundedness of $\mathcal{G}$ implies that also all auxiliary relations present in $\Upsilon_{\mathcal{S}}$ are bounded. We discuss each auxiliary relation separately. The artifact blocking relation R_{block} keeps a boolean flag for each artifact instance, so its cardinality depends on the number of instances in the model. Since the model is state-bounded, the number of artifact instances is bounded and so is R_{block}. The eligibility tracking table R_{exec} stores for each artifact instance a boolean vector describing the applicability of a certain PAC rule. Since the number of instances is bounded and so is the set of PAC rules, then the relation R_{exec} is also bounded. Similarly, one can show the boundedness of $R_{chg}^{m_i}$, $R_{chg}^{s_j}$ due to the fact that the number of stages and milestones is fixed a-priori. Let us now analyze internal message pools. By construction, $\mathcal{S}$ may contain at most one tuple in $R_{data}^{msg_k}$ and $R_{data}^{srv_p}$ for each artifact instance. This is enforced by the blocking mechanism R_{block}, which blocks the artifact instance at the beginning of a B-step and prevents the instance from injecting further events in internal pools. The outgoing message pool $R_{out}^{msg_q}$ may contain as much tuples per artifact instance as the amount of atomic stages in the model, which is still bounded. However, neither incoming nor outgoing messages are accumulated in the internal pool along the B-steps execution, since the final micro-step of the B-step is designed not to propagate any of the internal message pools to the next snapshot. Therefore, $\Upsilon_{\mathcal{S}}$ is state-bounded. □

From the combination of Theorems **??** and 1 and Lemma 4, we directly obtain:

Theorem 2. *Verification of μL_P properties over state-bounded GSM models is decidable, and can be reduced to finite-state model checking of propositional μ-calculus.*

Obviously, in order to guarantee verifiability of a given GSM model, we need to understand whether it is state-bounded or not. However, state-boundedness is a "semantic" condition, which is undecidable to check [9]. We mitigate this problem by isolating a class of GSM models that is guaranteed to be state-bounded. We show however that even very simple GSM models are not state-bounded, and thus we provide some modeling strategies to make any GSM model state-bounded.

Sufficient syntactic conditions have been studied in [9] to check whether the DCDS under study is state-bounded and, in turn, can be verified. However, when dealing with GSM these syntactic conditions become irrelevant, because

the resulting DCDS translations belong to a particular class of systems, for which studied syntactic conditions do not hold even in a trivial case. In particular, the way how immediate effect of an event is encoded, immediately makes the resulting DCDS violate the GR-acyclicity condition. Therefore, we have to find an alternative syntactic condition, working directly at the GSM level, in order to not only guarantee the verifiability of the model, but also provide feedback and explanation to the user. The following section defines such a condition.

6.1 GSM Models without Artifact Creation.

We investigate the case of GSM models that do not contain any *create-artifact-instance* tasks. Without loss of generality, we assimilate the creation of nested datatypes to the creation of new artifacts. From the formal point of view, we can in fact consider each nested datatype as a simple artifact with an empty lifecycle, and its own information model including a connection to its parent artifact.

Corollary 1. *Verification of μL_P properties over GSM models without* create-artifact-instance *tasks is decidable and can be reduced to finite-state model checking of propositional μ-calculus.*

Proof. Let $\mathcal{G}$ be a GSM model without *create-artifact-instance* tasks. At each stable snapshot Σ_k, $\mathcal{G}$ can either process an event representing an incoming one-way message, or the termination of a task. We claim that the only source of state-unboundedness can be caused by service calls return related to the termination of *create-artifact-instance* tasks. In fact, one-way incoming messages, as well as other service call returns, do not increase the size of the data stored in the GSM information model, because the payload of such messages just substitutes the values of the corresponding data attributes, according to the signature of the message. Similarly, by an inspection of the proof of Lemma 4, we know that across the micro-steps of a B-step, status attributes are modified but their size does not change. Furthermore, a bounded number of outgoing events could be accumulated in the message pools, but this information is then flushed at the end of the B-step, thus bringing the size of the overall information model back to the same size present at the beginning of the B-step. Therefore, without *create-artifact-instance* tasks, the size of the information model in each stable state is constant, and corresponds to the size of the initial information model. We can then apply Theorem 2 to get the result. □

6.2 Arbitrary GSM Models.

The types of models studied in paragraph above are quite restrictive, because they forbid the possibility of extending the number of artifacts during the execution of the system. On the other hand, as soon as this is allowed, even very simple GSM models, as the one shown in Fig. 4, may become state unbounded. In that example, the source of state unboundedness lies in the stage containing the

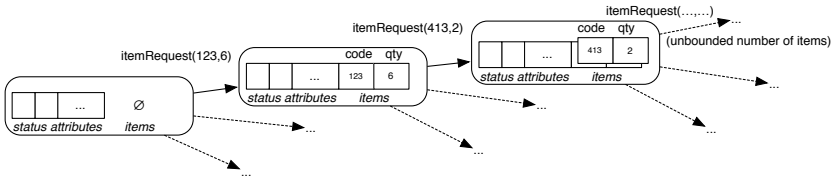

Fig. 4. Unbounded execution of the GSM model

"add item" task, which could be triggered an unbounded number of times due to continuous *itemRequest* incoming events. This, in turn, is caused by the fact that the modeler left the GSM model underspecified, without providing any hint about the maximum number of items that can be included in an order. To overcome this issue, we require the modeler to supply such information (stating, e.g., that each order is associated to at most 10 items). Technically, the GSM model under study has to be parameterized by an arbitrary but finite number N_{max}, which denotes the maximum number of artifact instances that can coexist in the same execution state. We call this kind of GSM model *instance bounded*. Two different policies may be adopted to provide N_{max}: *shared* vs *fixed distribution*. In the shared scenario, one global maximum value is fixed for the total number of artifact instances, regardless the artifact type. While this policy incorporates a flexible assignment of the available "slots", it does not guarantee any form of fairness about their allocation: they could all be occupied by instances of the same artifact type, blocking the possibility of creating instances of other artifact types. This problem can be overcome with the fixed distribution scenario, which requires the modeler to allocate available "slots" for each artifact type of the model, i.e. to specify a maximum number N_{A_i} for each artifact type A_i, then having $N_{max} = \sum_i N_{A_i}$. We discuss the execution of a GSM model with fixed distribution strategy.

In order to incorporate the artifact bounds into the execution semantics, we proceed as follows. First, we pre-populate the initial snapshot of the considered GSM instance with N_{max} blank artifact instances (respecting the relative proportion given by the local maximum numbers for each artifact type). We refer to one such blank artifact instance as *artifact container*. Along the system execution, each container may be: (i) filled with concrete data carried by an actual artifact instance of the corresponding type, or (i) flushed to the initial, blank state.

To this end, each artifact container is equipped with an auxiliary flag fr_i, which reflects its current state: fr_i is false when the container stores a concrete artifact instance, true otherwise. Then, the internal semantics of *create-artifact-instance* is changed so as to check the availability of a blank artifact container. In particular, when the corresponding service call is to be invoked with the new artifact instance data, the calling artifact instance selects the next available blank artifact container, sets its flag fr_i to $false$, and fills it with the payload of

$$R_{att}(id_R, fr, \overline{a}, \overline{s}, \overline{m}) \wedge fr = false \wedge s_p = true \wedge R_{block}(id_R, false) \wedge$$
$$\wedge\, R_{att}(id_Q, fr', \overline{a}', \overline{s}', \overline{m}') \wedge fr' = true \mapsto \quad (1)$$
$$a^Q_{create}(id_R, \overline{a}, id_Q) : \{ \quad (2)$$
$$R_{att}(id_Q, fr', \overline{a}', \overline{s}', \overline{m}') \rightsquigarrow \{R_{att}(id_Q, fr', \overline{\nu(a)}, \overline{s_0}', \overline{m_0}')[fr'/false]\} \quad (3)$$
$$R_{att}(id_Q, fr', \overline{a}', \overline{s}', \overline{m}') \rightsquigarrow \{R^{create}_{data}(id_R, id_Q)\} \quad (4)$$
$$R_{att}(id_R, \overline{a}, \overline{s}, \overline{m}) \rightsquigarrow \{R_{block}(id_R, true)\} \quad (5)$$
$$[\mathsf{SetupEligibilityTracking}], [\mathsf{CopyMessagePools}], [\mathsf{CopyRest}] \quad \} \quad (6)$$

Fig. 5. CA-rule encoding a create-artifact-instance service call

the service call. If all containers are occupied, the calling artifact instance waits until some container is released.

Symmetrically to artifact creation, the deletion procedure for an artifact instance is managed by turning the corresponding container flag fr_i to true. An example of a CA-rule, implementing the presented approach to artifact instance creation in DCDSs is presented in Figure 5, where: (1) represents a condition part of a CA-rule, ensuring the existence of a free container id_Q ($fr' = true$); (2) describes the action signature; (3) is an effect filling the container with a certain data ($\overline{\nu(a)}$) and marking it as occupied ($[fr'/false]$); (4) propagates an event denoting the artifact instance creation; (5) blocks the caller artifact instance to process such event; (6) are macros used respectively to set up eligibility tracking and to transport the unaffected data into the next snapshot.

We observe that, following this container-based realization strategy, the information model of an instance-bounded GSM model has a fixed size, which polinomially depends on the total maximum number N_{max}. The new implementation of *create-artifact-instance* does not really change the size of the information model, but just suitably changes its content. Therefore, Corollary 1 directly applies to instance-bounded GSM models, guaranteeing decidability of their verification. Finally, notice that infinitely many different artifact instances can be created and manipulated, provided that they do not accumulate in the same state (exceeding N_{max}).

References

1. Alessandro Armando and Serena Elisa Ponta. Model checking of security-sensitive business processes. In *Proc. of FAST*, volume 5983 of *LNCS*, pages 66–80. Springer, 2009.
2. Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. An abstraction technique for the verification of artifact-centric systems. In *Proc. of KR*. AAAI Press, 2012.
3. K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.

4. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. The MIT Press, 1999.
5. David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, 32(3), 2009.
6. Elio Damaggio, Richard Hull, and Roman Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. In *Proceedings of the 9th international conference on Business process management*, BPM'11, pages 396–412, Berlin, Heidelberg, 2011. Springer-Verlag.
7. Alin Deutsch, Richard Hull, Fabio Patrizi, and Victor Vianu. Automatic verification of data-centric business processes. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 252–267, New York, NY, USA, 2009. ACM.
8. Marlon Dumas. On the convergence of data and process engineering. In Johann Eder, Mária Bieliková, and A Min Tjoa, editors, *ADBIS*, volume 6909 of *LNCS*, pages 19–26. Springer, 2011.
9. Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, Alin Deutsch, and Marco Montali. Verification of relational data-centric dynamic systems with external services. *CoRR*, abs/1203.0024, 2012.
10. Babak Bagheri Hariri, Diego Calvanese, Giuseppe De Giacomo, and Riccardo De Masellis. Verification of conjunctive-query based semantic artifacts. In Riccardo Rosati, Sebastian Rudolph, and Michael Zakharyaschev, editors, *Description Logics*, volume 745 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.
11. Shoichi Morimoto. A survey of formal verification for business process modeling. In *Computational Science (ICCS 2008)*, volume 5102 of *LNCS*, pages 514–522. Springer, 2008.
12. A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3), 2003.
13. David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.
14. Frank Puhlmann and Mathias Weske. Using the *pi*-calculus for formalizing workflow patterns. In *Proceedings of the 3rd International Conference on Business Process Management*, volume 3649, pages 153–168, 2005.
15. Wil M. P. van der Aalst and Christian Stahl. *Modeling Business Processes - A Petri Net-Oriented Approach*. Springer, 2011.