Verifying GSM-based Business Artifacts

Pavel Gonzalez, Andreas Griesmayer, Alessio Lomuscio
Department of Computing, Imperial College London
{*pavel.gonzalez09, a.griesmayer, a.lomuscio*}*@imperial.ac.uk*

***Abstract*—Business artifacts allow to manage operations of business processes by capturing the key concepts and relevant information to guide their work flow. The Guard-Stage-Milestone (GSM) meta-model is a novel formalism for designing business artifacts that features declarative description of the intended behaviour without requiring an explicit specification of the control flow. Its concept of hierarchical structures of stages and explicit rules for the fulfilment of their guards and milestones supports the designing process but poses a challenge for formal verification. We show here how to approach the verification problem by developing a symbolic representation amenable to model checking. The feasibility of the approach is demonstrated by presenting a case study on the direct verification of a GSM model using a tool implementation.**

***Keywords*-Business Artifacts; Formal Verification; Model Checking**

I. Introduction

Business artifacts are a growing topic in web-services [1], [2], [3]. Differently from the standard process-based paradigm popular in web-services, in artifact systems data is given the same prominence as processes. One such approach, *Business Artifacts with Guard-Stage-Milestone Lifecycles* (GSM), was recently introduced as a declarative method for specification of artifact lifecycles [4], [5], [6]. The main advantage of this formalism is that it closely follows the intuitive way in which stakeholders think about their business. While much of the work is focused on the design, deployment and maintenance of GSM models, the verification of this formalism has not been tackled yet.

The key components of a business artifact are the *information* model that captures the data and the *lifecycle* model that controls the possible behaviour. A *GSM artifact system* consists of a set of artifact instances that communicate with the *environment* via *events*. Unlike most of the previous work on business artifacts, which modelled lifecycles as state machines, GSM allows for a declarative way of modelling using a hierarchical structure of *stages*. Each stage is equipped with a set of *guards* to control its activation, *milestones* to determine when its goals are achieved, and, optionally, sub-stages that provide direct support of parallelism within an instance. Guards and milestones are controlled by conditions that depend on data from the information model and are triggered by events. An important feature of the framework is that an occurrence of a single event may lead to a chain of changes in the artifact system, such as activation and inactivation of stages or achieving and invalidating milestones.

The declarative description of the model supports the natural way of thinking about the different stages of a working process and the necessary milestones towards achieving a certain goal. However, a complex artifact system supports a large number of services involving many stakeholders, making it difficult to asses whether the system will behave as intended once deployed. It is therefore desirable to have a mechanism in place to ensure the validity of the design. Of course, verification of services is an active field of research (e.g., see [7], [8]). The automatic verification of business artifacts was investigated in [9]; however, the lifecycles of artifacts were based on finite-state machines, which does not correspond to the GSM model. To the best of our knowledge, no solution currently exists for this declarative approach. This paper presents such a solution.

Specifically, we introduce a methodology to apply symbolic model checking [10], [11] on GSM and present an implementation, called GSM Checker (GSMC), that enables us to verify properties of models produced by Barcelona [5], a web-based engine developed by IBM Watson that supports the execution of GSM models. The verification is done directly on Barcelona models without the need for translating them into another modelling language. To verify the behaviour of an artifact system, we transform the GSM model into a finite-state machine and systematically examine all possible behaviours of the new model against specifications. The key aspect is the construction of a *transition relation* using rules that are derived from lifecycle models of artifacts.

We introduce GSM and symbolic model checking in Section II. Details on encoding of GSM, generation of the transition relation, and verification are given in Section III. The implementation is described in Section IV before we present a detailed case study in Section V. We conclude the paper and give some directions for future work in Section VI.

II. Preliminaries

We first present GSM and its semantics, followed by a brief introduction to model checking and the temporal logic CTL, which is used for the specification of properties.

This research was supported by the EU FP7 projects ACSI (FP7-ICT-257593) and DiVerMAS (FP7-PEOPLE-252184), and the EPRSC project EP/I00520X.

A. Business Artifacts with Guard-Stage-Milestone Lifecycles

This paper follows the formal definition of GSM presented in [5]. We define a *GSM model* Γ as a set of all artifact instances in the system and use the context variable x that ranges over the instances of artifact type R.

At the core of the *lifecycle model* is the notion of a *stage*, which consists of three following concepts. A *milestone* represents an operational objective that can be *achieved* or *invalidated* and corresponds to one of the ways in which a stage might reach completion. A *stage body* is a hierarchical cluster of activity intended to achieve a milestone, where each stage is *parent* of either a set of *sub-stages*, or a *task*. A stage becomes *inactive* when one of its milestones is achieved. A *guard* controls entry into the stage body, in which case the stage becomes *active*.

The *information model* keeps track of business relevant information in data attributes, as well as status attributes of stages and milestones. In particular, each stage S of an artifact instance x has associated a status variable $x.active_S$ that reflects if the stage is active or inactive. Similarly, $x.m$ reflects if milestone m is achieved. The communication between artifact instances and the environment is performed in form of *incoming* and *generated* events, which can be either a 1-way message, a 2-way service call, or an instance creation request. Generated events are created by tasks contained in *atomic stages*, i.e., stages without sub-stages. Both milestones and guards are controlled in a declarative manner by a condition $\chi(x)$ with a *triggering event* "***on*** $\xi(x)$" or an expression on data "***if*** $\varphi(x)$".

A *pre-snapshot* is an assignment to the variables in the information model, while a *snapshot* is a pre-snapshot that satisfies the following three *GSM Invariants*: all milestones of an active stage are false; all sub-stages of an inactive stage are inactive; at most one milestone of a stage can be achieved at any time.

The operational semantics of a GSM model Γ is based on the notion of a *Business step* (B-step), which corresponds to the impact of a single incoming event e on a snapshot Σ, and is considered the smallest unit of relevant change that occurs in the system. The impact of e is gradually constructed from 1) the *immediate effect* of the event, which can assign payload to data attributes and 2) a re-evaluation of the conditions in Γ by *Prerequisite-Antecedent-Consequent* (PAC) rules that can lead to changes in guards and milestones.

The *abstract* PAC rules are listed in Table I. Each PAC rule consists of the following three parts: the *prerequisite* (P) determines whether the rule is relevant to the previous snapshot Σ; the *antecedent* (A) contains a user-defined condition $\chi(x)$ and is evaluated relative to the next snapshot Σ'; the *consequent* (C) specifies the change to the value of a status attribute in the next snapshot Σ' if the rule is relevant and if A holds in Σ'. The first three PAC rules in the table are concerned with updating the status attributes on certain events, and the last three rules preserve invariants of the model. More specifically, PAC-1 governs activation of stage S if its guard $\chi(x)$ holds and its parent S^* is active; PAC-2 determines achieving milestone m if its corresponding stage S is active and its condition $\chi(x)$ holds; PAC-3 controls invalidating milestone m if it was achieved before and its invalidating condition $\chi(x)$ is true; PAC-4 directs invalidating milestone m when its corresponding stage S becomes active; PAC-5 governs inactivation of stage S when its milestone m is achieved; and PAC-6 induces inactivating of stage S when its parent S^* becomes inactive.

Table I
PAC RULE TEMPLATES.

Rule	Prerequisite	Antecedent	Consequent
PAC-1	$\neg x.active_S$	$\chi(x) \wedge x.active_{S^*}$	$+x.active_S$
PAC-2	$x.active_S$	$\chi(x)$	$+x.m$
PAC-3	$x.m$	$\chi(x)$	$-x.m$
PAC-4	$x.m$	**on** $+x.active_S$	$-x.m$
PAC-5	$x.active_S$	**on** $+x.m$	$-x.active_S$
PAC-6	$x.active_S$	**on** $-x.active_{S^*}$	$-x.active_S$

The incoming event e triggers a sequence of pre-snapshots $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$ with $\Sigma_o = \Sigma$, $\Sigma_1 = \mathit{ImmEffect}_e(\Sigma_0)$, and $\Sigma_n = \Sigma'$. The transition between pre-snapshots Σ_i and Σ_{i+1} is called a *micro-step*, whilst the B-step constitutes the transition from snapshot Σ to Σ'. The PAC rules are sequentially applied to Σ_i until a fixed-point is reached.[1] Each micro-step can generate an outgoing event if its associated atomic stage becomes active. These events are collected and sent to the environment in the last micro-step.

The *Toggle-once Principle*, that states that each status attribute can change its value at most once through the application of PAC rules, guarantees that the application of PAC rules terminates since there is a finite number of PAC rules. To ensure that the rules adhere to the principle, circular dependencies among PAC rules are not allowed, i.e., there must not be a set of rules where each C of a rule changes a status attribute required in A of another rule. A suitable order of the PAC rules is achieved via pre-determined topological sort of the *dependency graph* $DG(\Gamma)$ associated with the GSM model Γ. The graph contains nodes for all guards, stages and milestones for each artifact type R in Γ. The set of edges represents dependencies between individual nodes and it is based on ground PAC rules for Γ. If $DG(\Gamma)$ satisfies the acyclicity condition the model Γ is *well-formed*.

A web-based application that supports GSM models, called Barcelona, has been implemented by IBM Watson [5]. Users can model business artifacts with GSM lifecycles using a design editor and then immediately deploy these models on an execution engine. We present a motivating scenario modelled using Barcelona in Section V. More details on GSM can be found in [4], [5].

[1] PAC rules have three equivalent formulations [6]. We discuss only the *incremental* formulation, on which our approach is based.

B. Model Checking

Model checking [10] is an automated verification method that systematically explores all possible behaviour of a system under examination. The technology is widely used in verification, including the area of choreography and orchestration of web services [8]. It recently received much attention with the presentation of the 2007 Turing Award to E. M. Clarke, E. A. Emerson and J. Sifakis for their achievements in this area.

To apply model checking, we introduce the notion of a *state* s as a particular evaluation of the data and status attributes. Snapshots and pre-snapshots are both states in that sense. We will use the terms interchangeably when a further classification is not relevant or is clear from the context. The allowed changes of variable evaluations are captured by *transitions* between states.

Symbolic model checking [11] uses formulas to represent sets of states and their possible transitions in a *transition system* $M = \{S, \delta, I, AP\}$, where S is set of all possible states, $\delta \subseteq S \times S$ is the *transition relation* that captures all allowed transitions, $I \subseteq S$ is the set of *initial states*, and AP is a set of atomic propositions defined on the states. We write $\delta(s) = \{s' \mid (s, s') \in \delta\}$ to denote all *successors* of s. A run π of the system is a sequence of states $s_0, s_1, ...$ such that $s_0 \in I$ and $\forall_{i \geq 0} s_{i+1} \in \delta(s_i)$. We denote the i^{th} state of a run as $\pi[i]$, write $AP(s)$ for the propositions that hold at a given state, and use the temporal logic CTL [12] to specify properties on these propositions. CTL is a branching-time logic that allows to express properties about execution paths of a system. The syntax of a CTL formula φ is given as

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid EX\varphi \mid AG\varphi \mid E(\varphi U \varphi)$$

The semantics is defined inductively, where π_s denotes all runs starting from a set of states s. We say that a system M with s is a *model* of formula φ (given as $(M, s) \models \varphi$) if:

$$\begin{array}{ll}
(M,s) \models p & \textit{iff } p \in AP(s) \\
(M,s) \models \neg\varphi & \textit{iff } (M,s) \not\models \varphi \\
(M,s) \models \varphi_1 \wedge \varphi_2 & \textit{iff } (M,s) \models \varphi_1 \texttt{ and } (M,s) \models \varphi_2 \\
(M,s) \models EX\varphi & \textit{iff } \exists_{\pi \in \pi_s} : (M, \pi[1]) \models \varphi \\
(M,s) \models AG\varphi & \textit{iff } \forall_{\pi \in \pi_s} \forall_{i \geq 0} : (M, \pi[i]) \models \varphi \\
(M,s) \models E(\varphi U \psi) & \textit{iff } \exists_{\pi \in \pi_s} \exists_{k \geq 0} : (M, \pi[k]) \models \psi \wedge \\
 & \quad \forall_{j<k} (M, \pi(j)) \models \varphi
\end{array}$$

Additional operators can be constructed by combination of the ones given above (e.g., $EF\varphi := \neg AG \neg\varphi$, $\varphi \rightarrow \psi := \neg\varphi \vee \psi$). Intuitively, $X\varphi$, $G\varphi$, $F\varphi$, are path formulas that hold if φ evaluates to true in the next state, in all states, or eventually in some state of the path. Similarly, $\varphi U \psi$ holds if φ holds until ψ holds. The prefixes to path formulas A and E denote that a formula holds in a state if the formula holds for all paths (A) or at least one path (E) starting from the current state. A system M satisfies a formula φ if $(M, I) \models \varphi$.

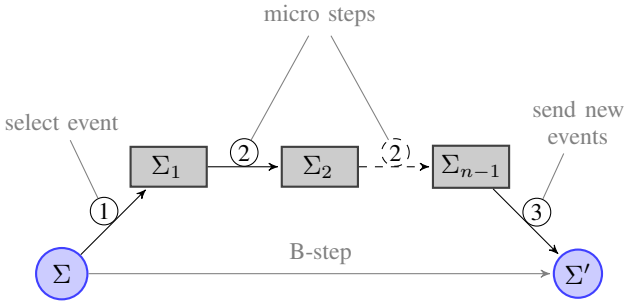

Figure 1. *B-steps* and *micro steps* of GSM.

Given a CTL formula, a model checker computes the states in which the formula holds. This can be done effectively using BDDs [11] as data structure to store states and transition relation.

III. Methodology

In the following we present a novel methodology to verify the behaviour of artifacts in terms of their possible sequences of B-steps. Note that GSM is not directly amenable to symbolic model checking as its semantics does not provide a transition system. Instead, a B-step is constructed from a number of micro-steps that apply changes in a sequence of updates until a fixed point is reached. Developing a transition relation from this declarative semantics requires further analysis of the process performing a B-step. We recall the three phases of the generation of a new snapshot in Figure 1:

- *process incoming events* (1): Each B-step processes one event that is selected from a set of pending events. The execution of the event may cause updates to the local data or perform structural actions like creating a new instance of an artifact.
- *application of PAC rules* (2): Effects from (1) may change the conditions of guards or milestones and thus trigger a PAC rule that updates the status attributes or issues events. These changes may lead to the execution of further PAC rules, resulting in a sequence of rule executions.
- *send new events* (3): To finalise a B-step, the executed event is removed from the set of pending events, and newly raised events from the PAC rule executions are added.

In the remainder of this section we show how to translate the GSM semantics to a transition system. We start with giving a suitable encoding for a state, followed by the definition of a transition relation from the three phases given above.

A. Encoding

To perform verification, we need to capture the current state of an artifact system with some additional information about

its environment. A snapshot Σ consists of status attributes and additional space for business data, which we support in form of Boolean, bounded integer and enumeration data types. In addition, a special status attribute "*exist*" for each artifact instance determines if it is active. This is used for obtaining a finite encoding by provisioning space for a bounded number of instances, which are activated upon reception of the *new* event. In the following we use a vector of variables $\overline{x}$ to subsume all artifact related data of a (pre-)snapshot along with possible user inputs for an event. A vector of variables $\overline{e}$ encodes the events that are pending at a certain snapshot Σ. For other pre-snapshots Σ_i only one $e \in \overline{e}$ is true and signifies the currently executed event.

We write $\overline{x}\,\overline{e}$ to denote the set of states spanned by the possible evaluations of the variables in these vectors. As the micro-steps operate on different states, we use three sets of variables to encode the transition relations: $\overline{x}\,\overline{e}$ for the previous snapshot Σ, $\overline{x}'\overline{e}'$ for the current pre-snapshot Σ_i, and $\overline{x}''\overline{e}''$ for the next pre-snapshot Σ_{i+1}. We use indices to access single elements in a vector.

B. Transition Relation

Using the encoding above, we build a symbolic transition relation $\delta \subseteq \Sigma \times \Sigma$ for a direct computation of B-steps. To this end, we build the transition relations $\delta_{①}$, $\delta_{②}$, and $\delta_{③}$ for the different phases from Figure 1 and concatenate them into $\delta = \delta_{①} \circ \delta_{②} \circ \delta_{③}$ to be able to compute $\delta(\Sigma)$ in a single step. To compute the successors for a set of states $\hat{\Sigma}$ with δ, we build the conjunction with the transition relation, remove the current state variables and replace them by the next states $\hat{\Sigma}' = \delta(\hat{\Sigma}) = (\exists_{\overline{x}\,\overline{e}} \hat{\Sigma} \wedge \delta)[\overline{x}\,\overline{e}/\overline{x}'\overline{e}']$. Concatenation of transition relations can be done similarly; but requires the introduction of an intermediate state and in our case the conversion between snapshots and pre-snapshots.

1) Execution of an Event: Phase ① generates the first pre-snapshot by picking a pending event e_i and assigning the result of its execution to the attributes x'. The effect of e_i is given by δ_{e_i} below, where $e_i = true$ states that e_i is pending in the snapshot, $e_i(\overline{x})$ returns the results of executing e_i using the current data, and all event variables in the resulting pre-snapshot except the one for event i are false:

$$\delta_{e_i} = \{\overline{x}\,\overline{e}\,\overline{x}'\overline{e}' \mid e_i = true \wedge \overline{x}' = e_i(\overline{x}) \wedge e_i' \bigwedge_{j \neq i} \neg e_j'\}$$

The event e_i changes attributes according to its type and the current values of $\overline{x}$, which also contains *user input*. The *new* event is executed by initialising the data of a corresponding artifact type and setting its *exist* flag to true. In a snapshot with several pending events one of them is selected non-deterministically. In the transition relation, this is expressed by disjunction of the possible choices:

$$\delta_{①} = \bigvee_{e_i \in \overline{e}} \delta_{e_i}$$

The resulting transition relation produces all possible initial pre-snapshots Σ_1 for any set of snapshots Σ.

2) Execution of the PAC Rules: The main challenge in computing the updates to status attributes within a B-step is the inter-dependency of the PAC rules. This leads to different sequences of pre-snapshots depending on the executed event and state of the system. The key property of the semantics that enables us to combine the different steps in ② into a single transition relation is the requirement that dependencies among PAC rules are not circular, i.e., that no *consequent* of a rule changes the variables needed in an *antecedent* of an earlier one. This allows us to find a single order of PAC rules that covers all permitted sequences of micro-steps. The transitions for PAC rules that are not applicable for a certain state are implemented such that all attributes are kept constant. The order is computed using the dependency graph $DG(\Gamma)$ before computing the transition relation.

To incorporate the changes in the pre-snapshot that are introduced by a PAC rule i, we have to take into account the prerequisite (P), the antecedent (A), and the consequent (C). The prerequisite checks a value on the last snapshot Σ. The antecedent is an expression over Σ and the *next* snapshot Σ'. Recall, though, that the ordering of the PAC rules ensures that none of the variables in A is changed during or after execution of the current rule, which allows us to operate on Σ and the pre-snapshot Σ_i. If A matches, the consequent updates the values for the next pre-snapshot while the values not touched by C remain as in Σ_i. The transitions generated by PAC rule i with the unprimed variables for Σ, primed for Σ_i and double primed for Σ_{i+1} are given as:

$$\begin{aligned}\delta_{r_i} = \{\overline{x}\,\overline{e}\,\overline{x}'\overline{e}'\overline{x}''\overline{e}'' \mid & P_{r_i}(\overline{x}) = true \wedge A_{r_i}(\overline{x}\,\overline{e}\,\overline{x}'\overline{e}') \wedge \\ & \overline{x}''\overline{e}'' = C_{r_i}(\overline{x}'\overline{e}') \\ & \vee P_{r_i}(\overline{x}) = false \wedge \overline{x}''\overline{e}'' = \overline{x}'\overline{e}'\}\end{aligned}$$

The transition relation gives a new pre-snapshot Σ_{i+1} for a given snapshot Σ and pre-snapshot Σ_i. If we build the conjunction of δ_{r_1} with $\delta_{①}$, we get a formula that describes the first and second pre-snapshots following any snapshot Σ. Because we are only interested in the latest pre-snapshot, we remove the middle state as follows:

$$\delta_{①} \circ \delta_{r_1} = \exists_{\overline{x}'\overline{e}'} \delta_{①} \wedge \delta_{r_1}[\overline{x}''\overline{e}''/\overline{x}'\overline{e}']$$

The result is again a formula in $\overline{x}\,\overline{e}$ and $\overline{x}'\overline{e}'$ and gives us the pre-snapshots that can be generated by Σ in two steps. To complete $\delta_{①} \circ \delta_{②}$ we repeat this step for all PAC rules in the pre-computed order.

Note that, while P only accesses a single variable, expressions for A are more complex and may contain specialised operators. For example, *StageActive(y)* is true if a status variable corresponding to the stage *y* is true, independently of whether it was activated during the current micro step

computation, or during some previous B-step. The truth value of this operator can be determined by accessing a status variable in $\overline{x}'$. By contrast, *StageActivatedOnEvent(y)* is only true if the stage just has been activated. Such an expression requires access to the previous snapshot Σ. If the corresponding flag was false there, and is true in $\overline{x}'$, then the stage was activated in the current B-step computation and the *StageActivatedOnEvent(y)* is true. Similarly, there is an expression that is true if and only if the current B-step computation was set off by event e_i, which requires access to $\overline{e}'$. Access to $\overline{e}$ allows us to reason about *pending* events.

3) Creating a new B-Step: The final phase ③ computes the resulting new snapshot Σ' from Σ and the last pre-snapshot Σ_n by computing a new set of pending events and holding the data from Σ_{n-1} constant. An event is pending if it either was pending before the last B-step but was not executed, or it was created in the last B-step. Computing the remaining events is simply done by selecting all events from Σ that are not in Σ_n:

$$\delta_{rem} = \{\overline{x}\,\overline{e}\,\overline{x}'\overline{e}'\overline{x}''\overline{e}'' \mid \overline{x}'' = \overline{x}' \wedge \bigwedge_{0 \le i < m} e_i'' = e_i \wedge \neg e_i'\}$$

An events is created when an atomic stage is activated during a B-step execution, i.e., the respective state attribute is set in Σ_{n-1} but not in Σ. For simplicity we denote the set of events that are issued by newly activated atomic stages as $\mathcal{E}$. The transition to issue the newly generated events is now given as:

$$\delta_{\mathcal{E}} = \{\overline{x}\,\overline{e}\,\overline{x}'\overline{e}'\overline{x}''\overline{e}'' \mid \overline{x}'' = \overline{x}' \wedge \bigwedge_{0 \le j < m} e_j'' = (e_j'' \in \mathcal{E} \vee e_j')\}$$

The final transition relation is now given as $\delta = \delta_{①} \circ \delta_{②} \circ \delta_{rem} \circ \delta_{\mathcal{E}}$ where the concatenation of δ_{rem} and $\delta_{\mathcal{E}}$ is done analogously to δ_{r_i}.

C. Verification

The transition relation computed above describes the behaviour of the system while executing events but does not create any incoming events from the environment. To fully check the system, we need to add these incoming events and cover every possible behaviour of the artifact system. To this end, we introduce a default agent that provides input to the artifact system. Intuitively, the role of this agent is to generate all possible interactions with the system that any arbitrary agent communicating with the system could trigger. This is done by allowing it to non-deterministically enable any event and user input before the artifact system reacts to these events. More formally, we give the transition relation of the agent δ_a as:

$$\delta_a = \{\overline{x}\,\overline{e}\,\overline{x}'\overline{e}' \mid \bigwedge_{0 \le i < m} e_i \rightarrow e_i' \wedge \overline{x}' = \mathit{chguser}(\overline{x})\}$$

where $\mathit{chguser}(\overline{x})$ sets arbitrary user input and keeps all other attributes in $\overline{x}$ constant.

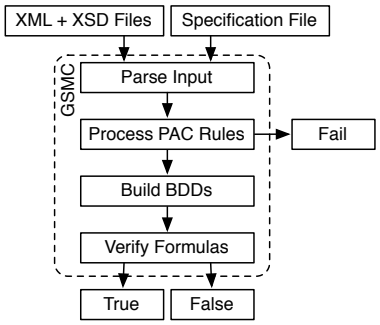

Figure 2. Architecture of GSMC.

When verifying the model, we consider all sequences consisting of alternating steps of agent and artifact system, starting from a single initial state s_0 with all data and events being zero.

IV. Implementation: GSMC

We have implemented the methodology in C++ using the CUDD library [13] for BDD operations. An important feature of GSMC is that it operates directly on Barcelona models designed in the Barcelona editor. A model of an artifact system is generated in XML format and can be deployed on the Barcelona engine after the verification. The properties are given as a plain text file containing formulas in the specification language. The output contains the result of the evaluation of the properties.

The internal architecture of GSMC is illustrated in Figure 2. The tool reads the inputs by using two parsers. The XML parser transforms the GSM model into an internal representation of the system, which consists of a set of objects representing instances of all artifact types in the system. The specification parser creates parse trees of the formulas.

Next, the pre-processor generates grounded PAC rules associated with the artifact system, constructs the dependency graph, and performs a topological sort on this graph. If the model is not a well-formed GSM model, the verification is halted and a cycle that violates the acyclicity condition is produced. After that, BDD variables are assigned to attributes and events and the BDD representations of PAC rules are constructed. The BDD for the transition relation of the artifact system and the agent are built following the procedure described in Section III.

Finally, the properties of the model are verified one by one. The BDD representation of the set of states in which a formula holds is computed using the transition relations. If the set contains the initial state then the formula is true in the model and false otherwise. GSMC implements standard algorithms to compute the set for CTL formulas [11].

The toolkit currently supports enumeration, bounded integer, and Boolean data types. All other types of data are coarsely abstracted as these. For example, string constants in

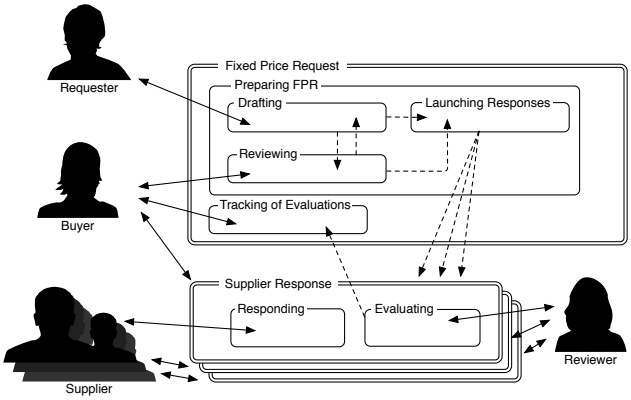

Figure 3. Overview of the Fixed Price scenario.

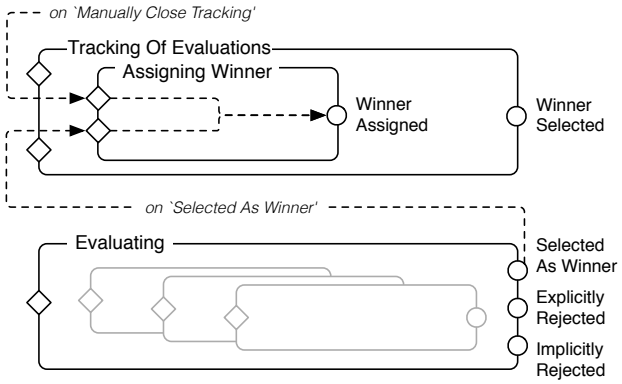

Figure 4. Structure of 'Evaluating' and 'Tracking Of Evaluations' stages.

the model are enumerated and can be compared for equality but concatenation is not supported. We also allow for at most one instance per artifact type in the system, attempts to create more result in the overflow flag being raised.

V. Case Study

A. Fixed Price Scenario

The Fixed Price contracting scenario [4] is based on a real-world application to facilitate purchasing of services or goods at fixed, predetermined prices. The application manages the interaction between a *requester* who wants to acquire a product, a *buyer* who manages the purchasing process, a number of optional *reviewers* who evaluate the order, and the actual *suppliers* of the product.

This scenario is modelled with two artifact types. Figure 3 gives an overview of the most important stages of the two types called Fixed Price Request (FPR) and Supplier Response (SR). An FPR instance is created when the requester makes the initial draft of the order in the 'Drafting' stage. Depending on the specifics, the buyer may initiate a reviewing process in the 'Reviewing' stage that may lead to redrafting. If the conditions are met, the 'Launching Responses' stage is activated where SR instances are automatically created. For each supplier, there is one SR instance, which manages the particular response. After a supplier responds to the request in the 'Responding' stage, the requester, the buyer, and possibly one or more reviewers evaluate the bid in the 'Evaluating' stage of the SR instance. The FPR instance manages these evaluations for each SR instance in the 'Tracking of Evaluations' stage and eventually checks with the buyer to select a winner who will be offered the contract for the order.

This is only a portion of the actual artifact system. The full model has 28 stages and 37 milestones in both artifact types together. We here evaluate the case where the order is sent just to a single supplier. However, we take into consideration the whole Barcelona model, which was supplied by IBM Watson. For more details we refer to Figures 4 and 5, where $\diamond$ represents guards, $\circ$ represents milestones, and the arrows represent dependencies between them. Labels on arrows express additional conditions on a guard or milestone.

B. Verification

We checked a number of properties directly on the Barcelona model of the Fixed Price scenario. We discuss four of the more interesting specifications below, where the formulas are slightly simplified to contain only relevant concepts from the stages explained above. The full model also contains, e.g., different styles of how a supplier can respond to the request. By using GSMC, we were able to identify two previously undiscovered bugs in the Barcelona model. We present the specifications that we found not to hold and explain why this is the case.

The first requirement concerns the reachability of milestones. It is reasonable to assume that all milestones of an artifact instance can be achieved at some point. This does not mean that all are achieved during a particular run but rather that at least one sequence of events leads to the achievement of any milestone. An unachievable milestone signifies that either the milestone is superfluous or that the system does not behave as expected. The following CTL formula specifies this requirement for the milestone 'Drafted' of the FPR artifact instance:

$$EF\ MilestoneAchieved('Drafted')$$

There are 37 milestones and by using GSMC we could verify that all the milestones, with the exception of 'Implicitly Rejected' milestone of the 'Evaluating' stage of the SR instance, can be achieved. The reason for this failure is that an SR instance becomes implicitly rejected only if another SR instance is selected as winner. This cannot happen in our model since we allow for at most one instance per artifact type.

The second specification, illustrated by Figure 4, says that whenever the 'Selected As Winner' milestone of the SR

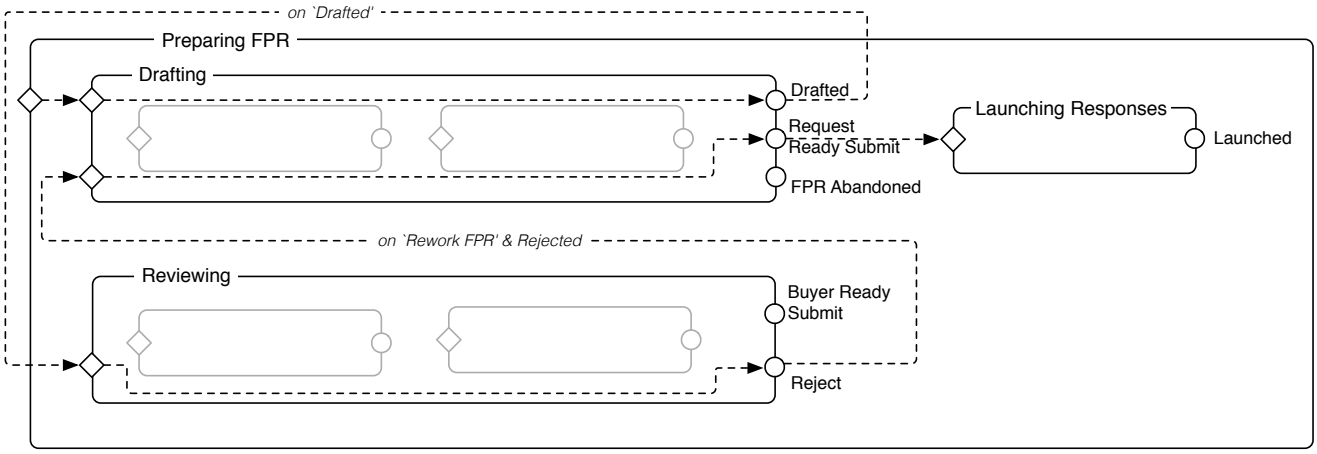

Figure 5. Structure of the 'Preparing FPR' stage.

instance is achieved, then the 'Winner Assigned' milestone of the FPR instance can be eventually achieved. This is formally specified as follows:

$$AG\ (MilestoneAchieved('SelectedAsWinner') \rightarrow EF\ MilestoneAchieved('WinnerAssigned')))$$

This formula holds in the model as expected. Note that we checked that the 'Winner Assigned' milestone *always can be achieved* (AG EF) rather than that it *always will be achieved* (AG AF). This is because 'Winner Assigned' requires interaction from the buyer to perform an 'Assign Winner' event. Since we check the artifact system for interactions with arbitrary agents, this event can be delayed forever. However, no matter what the user of the system does, the milestone can always be achieved when the event is executed.

The third specification we verified is similar to the previous one and is also illustrated by Figure 4. This time, though, we require that whenever the 'Winner Assigned' milestone of the FPR instance is achieved, then the 'Selected As Winner' milestone of the corresponding SR instance is achieved as well. In other words, the winner cannot be assigned within the FPR instance without the SR instance being selected as the winner. This property is expressed by the following CTL formula:

$$AG\ (MilestoneAchieved('WinnerAssigned') \rightarrow MilestoneAchieved('SelectedAsWinner'))$$

As it turns out, this formula is false in the model. This is because an agent can send a 'Manually Close Tracking' event to activate the 'Assigning Winner' stage manually. This causes the 'Winner Assigned' milestone to be reached without the milestone of the 'Evaluating' stage being updated. This means that the current implementation of the 'Evaluating' stage does not handle manual intervention of the agent properly. The problem can be fixed by adding another atomic sub-stage to the 'Tracking Of Evaluations' stage that deals with cancelling the tracking.

The last requirement we identify, shown in Figure 5, relates to the inner consistency of the 'Preparing FPR' stage. A requester may bypass the 'Reviewing' stage by sending an event to cause the 'Requester Ready Submit' milestone to be achieved, which activates the 'Launching Responses' stage directly. Since the latter sends the request to the suppliers, the 'Drafting' stage must not be reactivated since this would allow for changes to the already sent order. This can be specified as follows:

$$AG\ (MilestoneAchieved('RequesterReadySubmit') \rightarrow \neg EF\ StageActive('Drafting'))$$

We verified this formula using GSMC but found it to be false. Close inspection of the model shows that the problem occurs when a draft is reviewed and rejected at first, and then submitted without a second review via the 'Requester Ready Submit' milestone. In such a scenario, the second guard of the 'Drafting' stage, "*on 'Rework FPR' & 'Rejected'*", does not behave properly. Since the 'Rejected' milestone remains achieved from the initial review, an agent may activate the 'Drafting' stage at any time by sending the 'Rework FPR' event. Now the data attributes of the FPR instance can be significantly changed whilst the SR instances are being sent to suppliers. This may lead to unexpected consequences. The error can be corrected by changing the guards of the 'Drafting' stage such that they do not allow activation once the 'Launching Responses' stage is active.

The presented results demonstrate that GSMC provides invaluable assistance in modelling by identifying cases in which the system behaves as expected, and reporting cases that need more attention.

Table II
PERFORMANCE RESULTS.

Operation	Result	Memory	Time
Computation of Tr. Relation	✓	72MB	3.76s
Milestones can be Achieved	✓	100MB	129.21s
Winner can be Assigned	✓	112MB	26.52s
Winner Assignment Consistent	✗	84MB	9.10s
Order Consistent	✗	73MB	3.72s

C. Toolkit Evaluation

We conclude this section with a short discussion on the performance of GSMC. A pre-snapshot of the FP scenario is encoded by the model checker into BDD using 116 Boolean variables. Therefore, the state space of the model spans over approximately 8×10^{34} pre-snapshots. The construction of the transition relation, which is reused for the evaluation of all the specifications, requires three distinct sets of Boolean variables (348 in total).

We verified the properties on a 64-bit Fedora 16 Linux machine with a 3.47GHz Intel® Core™ i5 processor and 8GB RAM. Table II shows the results for the memory and CPU usage of the individual operations undertaken by GSMC. The first row reports the construction of the transition relation; the remaining rows give the performance for the properties verified in this section. Note, that the first specification consists of 37 separate formulas. These results suggest that the verification time and memory requirements of the tool are small even for a realistic scenario with a large model.

VI. CONCLUSION AND FUTURE WORK

We presented a methodology to model check declarative models of artifact systems by translating GSM artifact systems into a symbolic transition system used for symbolic model checking. A notable feature of our approach is that it is completely automatic. The GSMC toolkit takes files from the web-based GSM engine Barcelona as input. It has shown to be capable of handling large scenarios. We demonstrated the applicability using an example from a real-world application, proved the correctness of several properties, and identified two errors in the original model.

The tool has already proven to be very helpful for validating GSM models, but it is currently not sound or complete for general GSM models due to the limitations we impose on data types and the restriction to one instance per artifact type. For future versions of GSMC, we investigate the implementation of abstraction techniques [14] to improve data handling. We also work on the support of multiple instances per artifact type. A further interesting question is to check how an overall system may behave in presence of different agents. This gives rise to questions about the relationship between agents and the knowledge they have about the system and each other. Properties of this kind can be handled by extensions of CTL to epistemic logic [15].

REFERENCES

[1] D. Cohn and R. Hull, "Business artifacts: A data-centric approach to modeling business operations and processes," *IEEE Data Eng. Bull.*, vol. 32/3, 2009.

[2] N. Narendra, Y. Badr, P. Thiran, and Z. Maamar, "Towards a unified approach for business process modeling using context-based artifacts and web services," in *SCC '09*, 2009.

[3] D. Cohn, P. Dhoolia, F. T. Heath, F. Pinel, and J. Vergo, "Siena: From PowerPoint to web app in 5 minutes," in *ICSOC '08*, ser. LNCS, vol. 5364, 2008.

[4] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, and R. Vaculin, "Introducing the Guard-Stage-Milestone approach for specifying business entity lifecycles," in *WS-FM '10*, ser. LNCS, vol. 6551, 2011.

[5] R. Hull, E. Damaggio, R. D. Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, and R. Vaculin, "Business artifacts with Guard-Stage-Milestone lifecycles: Managing artifact interactions with conditions and events," in *DEBS '11*, 2011.

[6] E. Damaggio, R. Hull, and R. Vaculin, "On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles," in *BPM '11*, 2011.

[7] A. Lomuscio, H. Qu, and M. Solanki, "Towards verifying contract regulated service composition," in *ICWS '09*, 2008.

[8] W. Pacharoen, T. Aoki, A. Surarerks, and P.Bhattarakosol, "Conformance verification between web service choreography and implementation using learning and model checking," in *ICWS '11*, 2011.

[9] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu, "Automatic verification of data-centric business processes," in *ICDT '09*, 2009.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Inform. and Comput.*, vol. 98/2, 1990.

[12] E. Emerson and J. Y. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," *JCSS*, vol. 30/1, 1985.

[13] F. Somenzi, *CUDD: CU Decision Diagram Package - Release 2.5.0*, 2012. [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/

[14] F. Belardinelli, A. Lomuscio, and F. Patrizi, "An abstraction technique for the verification of artifact-centric systems," in *KR '12*, 2012, (To Appear).

[15] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: A model checker for the verification of multi-agent systems," in *CAV '09*, ser. LNCS, vol. 5643, 2009.