Model Checking GSM-Based Multi-Agent Systems*

Pavel Gonzalez[1], Andreas Griesmayer[2], Alessio Lomuscio[1]

[1] Department of Computing, Imperial College London
{pavel.gonzalez09, a.lomuscio}@imperial.ac.uk
[2] ARM, Cambridge
andreas.griesmayer@arm.com

Abstract. Artifact systems are a novel paradigm for implementing service oriented computing. Business artifacts include both data and process descriptions at interface level thereby providing more sophisticated and powerful service inter-operation capabilities. In this paper we put forward a technique for the practical verification of business artifacts in the context of multi-agent systems. We extend GSM, a modelling language for artifact systems, to multi-agent systems and map it into a variant of AC-MAS, a semantics for reasoning about artifact systems. We introduce a symbolic model checker for verifying GSM-based multi-agent systems. We evaluate the tool on a scenario from the service community.

1 Introduction

It has long been argued [1, 2] that agents are a fitting paradigm for service oriented computing (SOC). Indeed, agent-based research has contributed a wealth of techniques ranging from verification [3], protocols [4] and actual prototype implementations [5]. SOC is currently a fast moving research area with significant industrial involvement where highly scalable implementations play a key role. Agent-based solutions can shape developments in SOC if they remain anchored to emerging paradigms being put forward by the leading players in the area.

An increasingly popular paradigm being investigated in SOC is that of *business artifacts* [6]. In this approach *data*, not only processes, play a key part in the service description and implementations. While in traditional service composition processes are advertised at interface level, in the artifact approach both processes and the data structures are given equal prominence. Guard-Stage-Milestone (GSM) has recently been put forward [7] as a language for implementing business artifacts. GSM is a declarative language that provides a description of *stages*, which are clusters of activity pertaining to some artifact

* This research was supported by the EU FP7 project ACSI (FP7-ICT-257593). Andreas Griesmayer was partly supported by the Marie Curie Fellowship "DiVerMAS" (FP7-PEOPLE-252184) while at Imperial College London. Alessio Lomuscio acknowledges support from the UK EPSRC through the Leadership Fellowship grant "Trusted Autonomous Systems" (EP/I00520X/1).

data-structure. Stages are governed by *guards* controlling their activation and *milestones* determining whether or not the stage goals have been reached. The Guard-Stage-Milestone (GSM) approach to artifact systems [7] is particularly suitable for large unstructured processes where users have the freedom to decide what actions they perform and in what order. GSM is substantially influencing the emerging Case Management Modelling Notation standard [8]. IBM Watson developed `Barcelona`, a web-based application for modelling and execution of GSM-based artifact systems [7]. `Barcelona` provides a fully model-driven environment where a business operations model of an artifact system is created in a web-based design editor component, and then directly used for deployment on an execution engine.

While business artifacts are an attractive methodology for developing business processes and GSM-based services are a rapidly evolving area of research, they lack fully-fledged automatic methodologies for verification, orchestration and choreography. In this paper we put forward a technique and an implementation for the practical verification of business artifacts from a multi-agent system perspective. Specifically, we give a MAS-based formal model to GSM systems and define the model checking problem on this model. We observe the problem is undecidable in general, but note that as long as we can show the system operates within bounds, the problem is decidable. Within these parameters the methodology we report is sound and complete. We have built an implementation to verify automatically whether a GSM system, including a number of agents, satisfies given temporal-epistemic specifications which may include quantification over artifact instances. We test the technique against a noteworthy application developed by IBM.

Several contributions have so far studied the verification problem from a theoretical perspective [9–12]. The results obtained identify fragments of decidable settings either through restrictions on the specification language or the semantics. While these results are certainly valuable, they provide no methodology for the practical verification of GSM-based systems.

The work presented in this paper is based on [13] where `GSMC`, a model checker for GSM, is introduced. However, the semantics of the underlying formalism is one of plain transition systems and no support for agents in the system is provided. With no agents being present, no support is offered for views and windows, two key concepts that we fully support here. Additionally, as their concern is focused purely on the artifact system, the specification language only supports temporal logic, thereby making impossible to verify the information-theoretic properties of agents throughout an exchange as we do here.

2 The Guard-Stage-Milestone Artifact Model

Artifact systems form a conceptual basis for modelling and implementing business processes [6] and are given in terms of *artifact types*, which correspond to classes of key business entities. Each type has a *lifecycle model*, which describes the structure of the business process, and an *information model*, which gives an

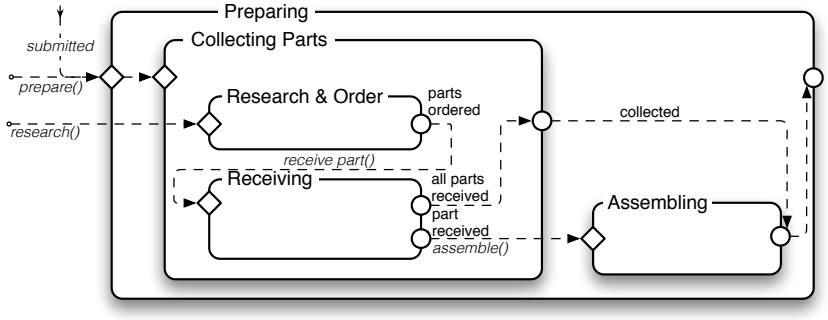

Fig. 1. A lifecycle model.

integrated view of the business data and the progress of the business process. The artifact system interacts with its environment via *events*. Our formal model of GSM is in line with [7].

GSM provides a declarative, hierarchical mechanism for specifying *lifecycle models*. Figure 1 gives a portion of the lifecycle of a manufacturing process and represents the core concepts: The boxes denote *stages*, which represent clusters of activity designed to achieve milestones ($\circ$) that represent operational objectives. A *guard* ($\diamond$) triggers activities in a stage when a certain condition is fulfilled. Stages are organised hierarchically, where the roots are called *top-level stages*, the leaves are called *atomic stages* and the non-leaf nodes are called *composite stages*. Atomic stages contain *tasks* that perform automated actions. Stages can run in parallel and own at least one milestone and one guard, while both milestones and guards belong to exactly one stage. A stage becomes *open* when one of its guards is fulfilled and *closed* when one of its milestones is *achieved*.

The example above gives the portion of the lifecycle of a manufacturing process that handles the procuring of the required building parts and the organisation of the assembly. When a new order is received by the manufacturer, the *submitted* event is sent to the artifact system, which triggers the guard of the *Preparing* stage, and in turn starts with *Collecting Parts*. When this stage is open, an employee of the manufacturer researches the required components and sends the *research* event to the artifact system which in turn processes the order of the required parts. When a part is received (event *part received*), the *Assembling* of the available parts is triggered; when *all parts* are *received* and *collected*, the *Preparing* stage can be closed. More details on this lifecycle will be discussed in Section 6.

Formally, an artifact system holds a number of *artifact instances* ι of *artifact type* $AT = \langle R, Att, Lcyc \rangle$, with R the *name* of the artifact type; Att the *information model* as set of *attributes*; and $Lcyc$ the *lifecycle model*. The *information model* Att is partitioned into the set Att_{data} of *data attributes* to hold business

data and the set Att_{status} of *status attributes* to capture the state of the lifecycle model. Each stage (resp. milestone), has a Boolean status attribute in Att_{status}, which is true iff the stage is *active* (resp. the milestone has been *achieved*). Both milestones and guards are controlled declaratively through *sentries*. A sentry of an artifact instance ι is an expression $\chi(\iota)$ in terms of incoming events and the status of the instance.

The progress of the lifecycle is driven by incoming events containing payloads, which are called *applicable* if the lifecycle is ready to consume them. An event with a specific payload is called a *typed external event*.

Definition 1 (Event Type). *An* event type *ET is a tuple $ET = \langle E, AT, A_1, \ldots, A_l \rangle$, where E is the* name *of the event type, AT is an artifact type, and $A_i \in Att_{data}$, where Att_{data} is the set of data attributes of AT.*

In addition, the opening of an atomic stage activates a task associated with the stage. It either performs an *automated system task*, such as the creation of a new instance, or corresponds to an operation outside the artifact system. Agents are not directly present in the GSM model, but it is assumed that human or artificial entities perform *tasks* and generate events for the system.

Definition 2 (GSM Model). *A* GSM model *Γ is a set of n artifact types AT_i for $1 \leq i \leq n$ and m event types ET_j for $1 \leq j \leq m$.*

Definition 3 (Snapshot of GSM Model). *A* pre-snapshot *of Γ is an assignment Σ that maps each attribute $A \in Att_\iota$ of each active artifact instance ι to an element in the domain of A. A* snapshot *of Γ is a pre-snapshot that satisfies the following GSM invariants: all sub-stages of a closed stage are closed; all milestones of an open stage are not achieved; at most one milestone of a stage can be achieved at any time.*

The operational semantics for GSM is based on the notion of a *business step* (B-step). This is an atomic unit that corresponds to the effect of processing one incoming event into the state of the artifact system. A B-step is computed by so called PAC rules which are formed from the sentries of the GSM model and has the form of a tuple $(\Sigma, e, \Sigma', Gen)$, where Σ, Σ' are snapshots, e is an incoming external event, and Gen is a set of outgoing external events generated by opening atomic stages during the B-step. For more details on the computation of a B-step please refer to [13].

3 Agent-Based GSM

Naturally, a GSM program only deals with the machinery related to the artifact system but does not provide a description of the agents interacting with it. To conduct the verification of agent-based GSM systems via model checking, we define A-GSM as an extension of GSM with a set of external agents.

The artifact system and agents communicate using events, where the available events for an agent depend on the current state. The system progresses by

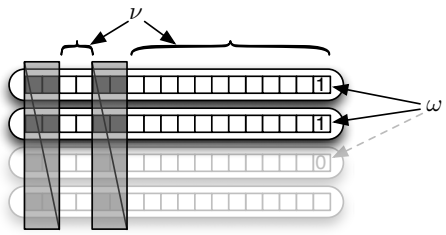

Fig. 2. Static and Dynamic visibility in A-GSM.

non-deterministically selecting an agent, which sends an event and triggers the execution of the AS. Selection of the event and execution of the AS are seen as one step, a stable state has no pending events.

3.1 Agent Description

Here we outline how the agents are specified and interact with GSM, thereby defining an *A-GSM instance*. The behaviour of an agent is determined by the permitted access to the artifact system AS and by local decisions regarding events to send. The former is determined by an agent's *role*, while the latter are defined for each agent individually.

The *role* is defined using the *view* ν for the visible attributes, the *window* ω to select the visible instances, and the *set of events* ϵ that are accepted by AS. While ν and ϵ are simple lists, $\omega_i(\iota)$ is a formula that is evaluated for a specific artifact instance ι and an agent i. The instance is exposed to the agent only if $\omega_i(\iota)$ evaluates to *true*. In addition to the role, the description of an agent also contains a *protocol* $\wp$ to determine its behaviour depending on the visible state of the AS, the agent's unique ID, and its private variables.

The concepts of ν, ω and ϵ are powerful tools to define the aspects agents can see and the ways they can interact with an artifact system. In Figure 2 the lines correspond to artifact instances that were created during run-time and the columns correspond to data attributes. ν defines a *static view* of the system, as it hides for each agent a fixed set of attributes depending on his role. For example, a *Customer* can only see that the state of an order moved from assembling to shipping, while a *Manufacturer* sees more detail, e.g., on suppliers. In contrast, ω gives a *dynamic* selection of the parts of the AS an agent can access in terms of the state of artifact instances as it hides complete instances depending on the current state. For instance, a *Manufacturer* may only see instances that represent unfinished orders while the window of a *Customer* can use the ID to restrict access to its own orders only.

Figure 3 gives an example of agent's description file. Visible data attributes are listed in the `view` field. The `window` field contains the formula for $\omega_i(\iota)$, where `$$` is a placeholder for the agent's ID. The field `instantiation` lists all

```
role Customer {
 view: CustomerId, ManufacturerId;
 window: CustomerId == $$;
 instantiation: CO;
 transformation: condense_stage(CO, Preparing);
};

agent Diogenes {
 role:  Customer;
 vars:  bool cancelled = false;
 protocol:
  Create_CO: CustomerId == "Diogenes" -> cancelled = cancelled,
  OnCancel: true -> cancelled = true;
};
```

Fig. 3. An agent definition file.

artifact types that agents of this role may instantiate; the corresponding instantiation events are added to ϵ. To specify the status attributes and events that are added to ν and ϵ, the field `transformation` holds a set of GSM operators that allow to hide parts of the GSM model Γ. Valid commands here are `hide_stage_status("S")` and `hide_milestone("m")` to hide the status attributes of stage S and milestone m respectively, and `delegate_sentry("s")` to remove events from ϵ_i if they are only used in sentry s. For convenience, the macro operators `condense_stage("S")` and `eliminate_stage("S")` hide all sub-stages or all information including guards and milestones respectively.

The private variables of an agent are defined in a list `var` of variable names $\overline{x}$ with their type and initial value. The `protocol` lists entries of the form `e :` γ `->` μ for all events e the agent can send. Multiple entries for the same event are treated as a disjunction. The condition γ is given in terms of data attributes of the instance ι, the payload, and the private variables. It defines the protocol function $\wp_i(\iota, \overline{x})$, which gives the set of events e with their respective payloads that can be sent in the current state. The protocol also gives an update function $\mu_i(e, \overline{x})$, which computes new assignments for the local variables depending on the selected event and the local state of the agent. By imposing conditions on the *payload* of an event e, $\wp$ also allows the agent to assign a specific value to its parameters, e.g., `CustomerId` is a parameter of `Create_CO`.

To handle *automated tasks*, we define an *AutoAgent*, which handles service calls and computations in the GSM model Γ and returns the result to the artifact system in form of an event. The *AutoAgent* holds pending tasks in a buffer t, has full access to Γ, and can send the return messages at any time, but is otherwise handled like any other agent.

4 Artifact-Centric Multi-Agent Systems

To analyse interactions within a GSM-based artifact system, we use artifact-centric multi-agent systems (AC-MAS) [10, 14], a semantics based on interpreted systems [15, 16]. As a GSM system supports multiple active artifact instances, we require a limited form of quantification. We therefore introduce IQ-CTLK, an extended version of CTLK, which is frequently used to describe agents that share a common environment. IQ-CTLK is a temporal-epistemic specification language with quantification over artifact instances. We give a formal mapping $f : \textit{A-GSM} \rightarrow \textit{AC-MAS}$, such that f preserves satisfaction of formulas in the specification language IQ-CTLK.

4.1 Formal Model

In an AC-MAS a set of agents $\mathcal{A}$ share an environment E constituted by the artifact system, i.e., the underlying elements of the environment are evolving artifacts of type R. The environment and an agent $i \in A$ have a local state (L_E and L_i respectively), where the agent can observe parts of the environment (i.e., some of the artifact instances in it). The local state of an agent thus comprises private data for the agent and observable aspects of the artifact system. We write $l_E(s)$ to represent the local state of the environment in the global state s, and $l_i(s)$ to represent the local state of agent i.

Definition 4 (Environment). *The* environment *represents an artifact system AS and is a tuple* $E = \langle L_E, Act_E, P_E \rangle$*, where* L_E *is the set of* local states*;* Act_E *is the set of* local actions*, which correspond to the interface of the AS; and* $P_E : L_E \rightarrow 2^{Act_E}$ *is the environment's* protocol function*, which enables actions to be executed depending on the local state of the AS.*

An agent is defined formally as:

Definition 5 (Agent). *An* agent *in an AS is a tuple* $i = \langle L_i, Act_i, P_i \rangle$*, where* L_i *is the set of* local states *including the observable aspect of the AS;* Act_i *is the set of* local actions *corresponding to events that can be sent by the agent onto the AS and including an action skip for performing a null action; and* $P_i : L_i \rightarrow 2^{Act_i}$ *is the local* protocol function*.*

An agent i and the environment E communicate by synchronisation on actions, where Act_E corresponds to events enabled by the artifact system, and $Act_i \subseteq Act_E \cup \{skip\}$ is the set of *local actions* corresponding to events that can be executed by the agent and the idle action *skip*. Given the relation between notions of *action* in interpreted systems and *event* in GSM, we use these terms interchangeably in the rest of the paper. As in plain interpreted systems, protocols are used to select the actions performed in a given state.

Following the terminology of [14] we define an AC-MAS as the composition of the environment and a number of agents as follows:

Definition 6 (AC-MAS). *Given an environment E and a set of agents $\mathcal{A}$, an* artifact-centric multi-agent system *is a tuple $\mathcal{P} = \langle S, \mathcal{I}, \tau \rangle$, where $S \subseteq L_E \times L_1 \times \cdots \times L_n$ is the set of* reachable global states*; $\mathcal{I}$ is the* initial state*; and $\tau : S \times Act \to 2^S$ with $Act = Act_E \times Act_1 \times \cdots \times Act_n$ is the* global transition relation*. The transition $\tau(s, \alpha)$ is defined for $\alpha = \langle a_E, a_1, \ldots, a_n \rangle$ iff $a_E \in P_E(l_E(s))$, and $\exists_{0 \leq i < n} : a_i \in P_i(l_i(s))$, $a_E = a_i \wedge \forall_{j \neq i} : a_j = skip$.*

Intuitively, the conditions on the transition relation limit the communication between agents and environment such that environment and agent agree on the same action. The environment enables actions when the artifact system is ready to consume them, while the agent i decides on the actions to execute depending on a local strategy encoded in P_i. Only one agent can interact with the environment at a time while the others are idle.

We write $s \to s'$ iff there exists an action α, such that $s' \in \tau(s, \alpha)$, and call s' the *successor* of s. A *run* r from s is an infinite sequence $s^0 \to s^1 \to \ldots$ with $s^0 = s$. We write $r[i]$ for the i-th state in the run and r_s for the set of all runs starting from s. A state s' is *reachable* from s if there is a run from s that contains s'. In line with the semantics of epistemic logic [16], we say that the states s and s' are *epistemically indistinguishable* for agent i, or $\sim_i$, iff $l_i(s) = l_i(s')$.

4.2 The Logic IQ-CTLK

We are interested in specifying temporal-epistemic properties of agents interacting with the artifact system, as well as the system itself. Since GSM supports the dynamic creation of unnamed artifacts, the properties need to be independent of the actual number or possible IDs of artifact instances in the system. To specify such properties we here define a temporal-epistemic logic that supports quantification over the artifact instances. We call the logic IQ-CTLK, for *Instance Quantified CTLK*, where CTLK is the usual epistemic logic on branching time. It is a subset of FO-CTLK where quantification can only be over artifact instances but not data. The syntax is defined in BNF notation as follows:

$$\begin{aligned} \varphi ::= & \, p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi) \\ & \mid K_i\varphi \mid \forall x : R \; \varphi \mid \exists x : R \; \varphi \end{aligned}$$

where R is the name of an artifact type and p is an atomic proposition over the agents' private data and the attributes of active instances that are specified in terms of *instance variables* bound by the quantification operators. The quantified instance variables range over the active instances of a given artifact type R in the state where the quantification is evaluated and must be bound. We write $R(s)$ for the set of instances of type R in s.

The defined operators are read as follows: $EX\varphi$ means *there is a next state in which φ holds*; $EG\varphi$ conveys *there is a run where φ holds in every state*; $E(\varphi U \psi)$ denotes *there is a run in which φ holds until ψ holds*; $K_i\varphi$: expresses *agent i knows φ*; $\forall x : R$ represents *for all instances of type R*; and $\exists x : R$ says *there is an instance of type R*. The remaining CTL operators can be constructed by

combination of the ones given above in the standard way. For example, $AG\ \forall x : OrderAF\ K_i x.sent$ encodes the property expressing that in any reachable state, agent i will eventually know that the attribute *sent* is set to true for every active instance of type *Order*.

We inductively define the semantics of IQ-CTLK over an AC-MAS $\mathcal{P}$ as follows. A formula φ is true in a state s of $\mathcal{P}$, written $(\mathcal{P}, s) \models \varphi$, iff:

$$
\begin{array}{ll}
(\mathcal{P}, s) \models p & \textit{iff } p \in s \\
(\mathcal{P}, s) \models \neg\varphi & \textit{iff it is not the case that} (\mathcal{P}, s) \models \varphi \\
(\mathcal{P}, s) \models \varphi_1 \vee \varphi_2 & \textit{iff } (\mathcal{P}, s) \models \varphi_1 \textit{ or } (\mathcal{P}, s) \models \varphi_2 \\
(\mathcal{P}, s) \models EX\varphi & \textit{iff } \exists_{s'} : s \to s' \textit{ and } (\mathcal{P}, s') \models \varphi \\
(\mathcal{P}, s) \models EG\varphi & \textit{iff } \exists_{r \in r_s} : \forall_{i \geq 0} : (\mathcal{P}, r[i]) \models \varphi \\
(\mathcal{P}, s) \models E(\varphi U \psi) & \textit{iff } \exists_{r \in r_s} : \exists_{k \geq 0} : (\mathcal{P}, r[k]) \models \psi \textit{ and} \\
 & \quad \forall_{j<k} (\mathcal{P}, r[j]) \models \varphi \\
(\mathcal{P}, s) \models K_i\varphi & \textit{iff } \forall s' \in S : s \sim_i s' \textit{ implies } (\mathcal{P}, s') \models \varphi \\
(\mathcal{P}, s) \models \forall x : R\ \varphi & \textit{iff } \forall u \in R(s) : (\mathcal{P}, s) \models \varphi[u/x] \\
(\mathcal{P}, s) \models \exists x : R\ \varphi & \textit{iff } \exists u \in R(s) : (\mathcal{P}, s) \models \varphi[u/x]
\end{array}
$$

The above semantics provides an information-theoretic definition of knowledge, i.e., K_i expresses what agent i can infer from the information available to him. An agent *knows* that φ is true in state s if φ is true in all states s', which the agent cannot distinguish from s. Finally, given an AC-MAS model $\mathcal{P}$ and an IQ-CTLK specification φ, the model checking problem concerns the decision as to whether the formula φ holds at the initial state of $\mathcal{P}$.

Note that the above semantics provides an information-theoretic definition of knowledge, i.e., K_i expresses what agent i can infer from the information available to him. An agent *knows* that φ is true in state s if φ is true in all states s', which the agent cannot distinguish from s. This means the agent does not need to build a knowledge base, from which he can deduce new information, since he already knows everything he could possibly deduce in a certain situation.

Given an AC-MAS model $\mathcal{P}$ and an IQ-CTLK specification φ, the model checking problem concerns establishing whether the formula φ holds at the initial state of $\mathcal{P}$, written $\mathcal{P} \models \varphi$. In the context of our formal model, an AC-MAS $\mathcal{P}$ satisfies φ if $(\mathcal{P}, \mathcal{I}) \models \varphi$. Intuitively this means that the model $\mathcal{P}$ satisfies φ if φ is true in the initial state of $\mathcal{P}$.

This was shown to be undecidable on similar semantic structures and more expressive logics [11]. In the following sections, we will achieve decidability by bounding the data and the number of instances present. We will also show the implementation of the technique to demonstrate its feasibility.

4.3 Mapping to Agent-Based GSM to AC-MAS

We now establish the formal mapping f : *A-GSM* $\to$ *AC-MAS*. Note that the semantics for the local states and protocols of agents in A-GSM are given

in terms of AC-MAS. We define the map by constructing the environment $\langle L_E, Act_E, P_E\rangle$ from the GSM model Γ of a given artifact system and create an agent $\langle L_0, Act_0, P_0\rangle$ for the *AutoAgent*, and $\langle L_i, Act_i, P_i\rangle$ with $1 \leq i \leq n$ for each external A-GSM agent. We identify a GSM event e with an AC-MAS action a and will omit the conversion in the following for ease of presentation. The sets of actions Act_E, Act_0, and Act_i are thus directly defined by the events the AS provides and the permissions of the agents.

Global state: To construct a global AC-MAS state $\langle l_E, l_0, \ldots, l_n\rangle \in S$ from an snapshot Σ, an *AutoAgent* buffer t and the local agent states x_i, we identify l_E with Σ and l_0 with t. The local states $l_1, \ldots, l_n$ of the external agent comprise the state of the private variables x_i and the *projections* $\Sigma_{|i}$ of the environment snapshot such that:

$$\Sigma_{|i} = \{\iota \mid \exists_{\iota' \in \Sigma} : \omega_i(\iota') \wedge \iota = \iota'_{|\nu_i}\}$$

where $\iota'_{|\nu_i}$ is the restriction of the artifact instance ι' to the variables in ν_i (variables not in ν_i are replaced by $\perp$).

The initial state $\mathcal{I}$ is the empty state without any artifact instances in Σ or pending tasks in l_0. Private variables are initialised to their initial value.

Protocol: By construction, GSM executes only *applicable* events and blocks all others. Artifact instantiation events are always permitted. This is reflected in the environment protocol P_E:

$$P_E(\Sigma) = \{a \mid \exists_{\iota \in \Sigma} : (\chi \in X(\Gamma) \wedge \chi(\iota, a)) \vee a \in inst\}$$

where $X(\Gamma)$ is the set of all sentries in the milestones and guards of Γ and $\chi(\iota, a)$ is the evaluation of a sentry χ with respect to the action a and status attributes $Att_{status} \in \iota$. We write *inst* for the set of artifact instantiation events. The *AutoAgent* stores the set of pending tasks in its buffer t and sends them at a later point to Γ. Thus, the protocol simply selects any pending task from its buffer by using the expression $P_0(t) = \{a | a \in t\}$. The protocol of an agent i gives the set of actions that are available in visible instances of its local state and satisfy its local protocol:

$$P_i(l_i) = \{a \mid \exists_{\iota \in l_i} : a \in \epsilon_i(\iota) \cap \wp_i(\iota, x_i)\}$$

These components suffice to instantiate a full AC-MAS from Definition 6. With these details in place we conclude the formal map from *A-GSM* to *AC-MAS*. In the remainder of the paper we present an implementation of a model checker for *IQ-CTLK* on *AC-MAS*.

5 Implementation

To perform AC-MAS model checking, we have extended `GSMC` [13] model checking. The new version, numbered 0.8.5[3], is written in C++ and uses the CUDD

[3] The pre-compiled binaries of the tool can be downloaded from `http://www.doc.ic.ac.uk/~pg809/gsmc/0.8.5.tar.gz`

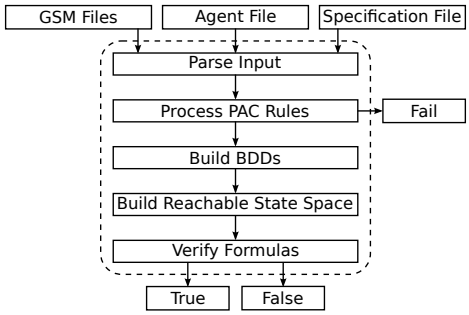

Fig. 4. Architecture of `GSMC`.

library [17] for the back-end symbolic computations. `GSMC` builds the model and the transition relation and performs a symbolic state space exploration based on BDDs. The GSM model and the specification of the *AutoAgent* are directly loaded from the `Barcelona` XML input file; agent definitions are given in form of a configuration file as shown in Figure 3. The internal architecture of the model checker is illustrated in Figure 4.

To obtain finite state models, we introduce a *bound* on the number of instances that can be generated and use abstraction to create finite data; an *overflow* flag indicates if the bound was reached during a run. We allocate BDD variables for the states of the agents and the maximum number of artifact instances present in a run. The basic layout of the BDD data structure is shown in Figure 5. We introduce an *Overflow* flag that indicates if the number of instances or data values were exceeded in a run. We pay special attention to this case because some of the results of the check may be unsound and require a re-check with higher bounds. We also capture the *Event ID* and *Payload* of the next action a that is to be executed. The artifact instances correspond to Σ. The actual number and size of these fields depend on the artifact type and the bounds that are fixed at the start of the verification. The special flag *Created* in each artifact instance indicates whether it was instantiated in the corresponding run. The task buffer fields t with a *Pending* flag and the corresponding payload belong conceptually to the *AutoAgent*, but are stored in the artifact state space for technical reasons. Private variables of agents complete the data structure.

Any IQ-CTLK formula φ to be verified is first rewritten by replacing the quantification operators with formulas that range over the actual instances. However, because artifact instances are created dynamically at run-time, the number of *active* instances is not known *a priori* and needs to be considered in the formula. We use the expression $created(\iota)$ to check if an instance was created

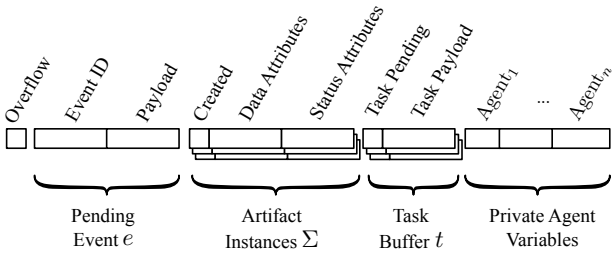

Fig. 5. Layout of the BDD data structure.

(the *Created* flag is set) and rewrite the quantified formulas as follows:

$$\forall x : \varphi \Rightarrow \bigwedge_{\iota \in \Gamma} : created(\iota) \rightarrow \varphi$$
$$\exists x : \varphi \Rightarrow \bigvee_{\iota \in \Gamma} : created(\iota) \wedge \varphi$$

Note that, for any existential formula to be valid, at least one of the artifact instances needs to be active; this is not the case in the initial state because no artifact instance has been created yet. Quantifiers can be arbitrarily nested and are resolved recursively. Once the details above are considered, `GSMC` follows existing methodologies to perform the verification of temporal-epistemic formulas [18].

5.1 Limitations

The bound in the number of instances restricts the possible behaviour of the system, while data abstraction leads to an over approximation. This may lead to loss of soundness or completeness when the limit of artifact instances is reached. The exact outcome depends on the type of the property considered. A violation of a universal property, for instance, does denote a violation on the full unbounded model even if the bound was exceeded during the computation. If an existential property is not satisfied, no conclusion can be drawn regarding the full model in general. These are limitations in the technique at present but, as we show in the following, interesting scenarios can still be analysed.

6 Experimental Results

We evaluated `GSMC` on the Order-to-Cash scenario, a simplified version of the IBM back-end order management application supplied by IBM Research [7]. In this scenario a manufacturer schedules the assembly of a product based on a confirmed purchase order from a customer. Typically, a product requires several

Table 1. Properties of the Order-To-Cash case study.

$$AG\ \forall x : CO((x.BId = Dio \wedge \neg x.Cancelled) \rightarrow K_{Dio}\ EF\ x.Received) \quad (1)$$
$$EF\ \exists x : CO(x.BId \neq Dio \wedge K_{Dio}\ x.Received) \quad (2)$$
$$AG\ \forall x : CO((x.BId = Dio \wedge x.Ready) \rightarrow K_{Dio} x.Parts = 3) \quad (3)$$
$$EF\ \exists x : CO(x.BId = Dio \wedge x.Cancelled \wedge \neg Dio.cancelled) \quad (4)$$

components that are sourced from different suppliers. After all components have been delivered the product is assembled and shipped to the customer.

The GSM program is specified in the form of a single-artifact `Barcelona` schema consisting of 9 stages and 11 milestones. To verify the model we performed small modifications to abstract from concrete products and created three agent roles for the above scenario: 1) a *Customer* who creates an artifact instance that represents the order and can only see instances they created; 2) a *Manufacturer* who fulfils the order and can see only uncompleted instances of orders sent to him by a customer; and 3) a *Carrier* who ships the finished product to the customer, and who can see only instances of orders that are to be shipped via them.

Figure 1 gives the lifecycle of the *Preparing* stage. It is controlled solely by the manufacturer, who, upon receiving the order, launches a research process to identify suitable suppliers and orders the required components. The assembling process can begin when the first component is received and remains active until all the components are collected. This is modelled by introducing a counter; the process is considered complete when 3 components have arrived.

Table 1 reports the properties we checked for different numbers of agents and artifact instances, where *Dio* is a customer agent (*Diogenes*) and *CO* stands for the *CustomerOrder* artifact type. Property (1) represents that *Diogenes* knows that, unless he cancels an order, the product can always be received in all of his orders. (i.e., that there is no deadlock in processing an order: An order can always be delivered or is cancelled). To check that the order is private to the customer, property (2) expresses that *Diogenes* may know a product is received for an order with different owner. Property (3) encodes the ability of an agent to deduce information it can not directly observe by checking if *Diogenes* always knows there are 3 *Parts* collected in all of his orders when the milestone *Ready* is achieved. Property (4) implies that an agent other than *Diogenes* can cancel an order that belongs to *Diogenes*. This is done by using a private variable, which is set true only if *Diogenes* executed the *Cancelled* event.

We ran the tests on a 64-bit Fedora 17 Linux machine with a 2.10GHz Intel Core i7 processor and 4GB RAM and measured the number of reachable states, memory used, and CPU time required. The model checker evaluated the properties (1) and (3) to be true and the properties (2) and (4) to be false in the model.

Table 2. Reachable states, memory and time usage for different numbers of artifact instances ι and agents.

	3 agents			15 agents			
$\#\iota$	#states	MB	s	#states	MB	s	
1	1.17 e2	27	0.1	2.92 e3	31	0.2	
2	3.71 e3	52	0.7	4.16 e6	70	4.9	
3	1.16 e5	64	5.9	5.82 e9	84	65.5	
4	3.67 e6	96	42.1	8.01 e12	222	360.2	
5	1.18 e8	195	176.7	1.09 e16	539	1419.6	

This is in line with our intuition of the model and shows that the GSM program of Order-to-Cash application is indeed correct with respect to the requirements.

Table 2 reports the performance for 3 agents (one for each role) and 15 agents respectively (6 customers, 5 manufacturers, and 4 carriers). We see that the run-time grows exponentially with the number of artifact instances, while the number of agents influences the resource usage only moderately. This is because additional agents add fewer states than additional artifact instances. The results show that the tool has the ability to effectively handle large state spaces, which is required to model realistic artifact systems with complex agent interactions.

7 Conclusions

In this paper we put forward a technique for the practical verification of GSM-based MAS. The approach consists of defining a formal map from the declarative, executable language GSM to an extension of previously studied artifact-centric MAS, a semantics for reasoning about MAS in a quantified setting of the artifact system environment. We reported on a fully-fledged model checker that implements this formal map and supports temporal-epistemic specifications in which quantification is allowed over artifact instances. The experimental results obtained against the Order-to-Cash application led us to conclude that the practical verification of reasonably sophisticated GSM-based MAS is feasible and scalable in valuable scenarios in business processes and services. However, GSM and Barcelona are still a topic of active research and development and sophisticated and stable models are hard to come by.

We plan to extend the work reported here in a number of ways, including the support of limited quantification over data. Theoretical studies [10, 14] point to high-undecidability in settings where unbounded data is present. For this reason we will work on existential abstraction and data abstraction to achieve a transfer of the verification outcome from abstract to concrete models. In particular we work on 3 valued abstraction [19], an abstraction technique that supports the detection of insufficient information in the abstraction.

References

1. Singh, M., Rao, A.S., Georgeff, M.: Formal methods in DAI: Logic-based representation and reasoning. In Weiß, G., ed.: Multiagent Systems: A Modern Approach to Distributed Artifical Intelligence. MIT Press (1999) 331–376
2. Bultan, T., Su, J., Fu, X.: Analyzing conversations of web services. IEEE Internet Computing **10**(1) (2006) 18–25
3. Lomuscio, A., Qu, H., Solanki, M.: Towards verifying contract regulated service composition. Autonomous Agents and Multi-Agent Systems **24(3)** (2012) 345–373
4. Singh, M.P., Huhns, M.N.: Service-oriented computing - semantics, processes, agents. Wiley (2005)
5. Baldoni, M., Baroglio, C., Mascardi, V.: Special issue: Agents, web services and ontologies: Integrated methodologies. Multiagent and Grid Systems **6**(2) (2010) 103–104
6. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **32**(3) (2009) 3–9
7. Hull, R., Damaggio, E., De Masellis, R., et al.: Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: Proceedings of the International Conference on Distributed Event-Based Systems (DEBS'11). (2011) 51–62
8. Object Management Group: Proposal for: Case management modeling and notation (CMMN) specification 1.0 (February 2012) Document bmi/12-02-09.
9. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web applications. Journal of Computer and System Sciences **73**(3) (2007) 442–474
10. Hariri, B.B., Calvanese, D., Giacomo, G.D., Deutsch, A., Montali, M.: Verification of relational data-centric dynamic systems with external services. CoRR (2012)
11. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of deployed artifact systems via data abstraction. In: ICSOC'11. (2011) 142–156
12. Belardinelli, F., Lomuscio, A., Patrizi, F.: Verification of GSM-based artifact-centric systems through finite abstraction. In: ICSOC'12. (2012) 17–31
13. Gonzalez, P., Griesmayer, A., Lomuscio, A.: Verifying GSM-based business artifacts. In: Proceedings of ICWS'12. (2012) 25–32
14. Belardinelli, F., Lomuscio, A., Patrizi, F.: An abstraction technique for the verification of artifact-centric systems. In: Proceedings of Principles of Knowledge Representation and Reasoning (KR'12). (2012) 319–328
15. Parikh, R., Ramanujam, R.: Distributed processes and the logic of knowledge. In: Logic of Programs. Volume 193 of LNCS. (1985) 256–268
16. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. The MIT Press (1995)
17. Somenzi, F.: CUDD: CU decision diagram package - release 2.5.0. `http://vlsi.colorado.edu/~fabio/CUDD/` (2012) January 2013.
18. Lomuscio, A., Qu, H., Raimondi, F.: Mcmas: A model checker for the verification of multi-agent systems. In: Proceedings of Computer Aided Verification (CAV'09). Volume 5643 of LNCS. (2009) 682–688
19. Shoham, S., Grumberg, O.: 3-valued abstraction: More precision at less cost. Information and Computation **206**(11) (2008) 1313 – 1333