# ACSI – Artifact-Centric Service Interoperation

SEVENTH FRAMEWORK PROGRAMME

# D2.2.2 Model Checking Tool for Artifact Interoperations (MOCAI) – Iteration II

| | | |
|---|---|---|
| Project Acronym | ACSI | |
| Project Title | Artifact-Centric Service Interoperation | |
| Project Number | 257593 | |
| Workpackage | 2 | Formal-based techniques and tools |
| Lead Beneficiary | Imperial College of Science, Technology and Medicine | |
| Editors | Pavel Gonzalez | Imperial |
| | Andreas Griesmayer | Imperial |
| Contributors | Francesco Belardinelli | Imperial |
| | Alessio Lomuscio | Imperial |
| | Fabio Patrizi | UoR |
| Reviewers | Dirk Fahland | TU/e |
| | Giuseppe De Giacomo | UoR |
| | Marco Montali | Bolzano |
| Dissemination Level | Public | |
| Contractual Delivery Date | 01/06/2012 | |
| Actual Delivery Date | | |
| Version | 2.0 | |

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 257593

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 257593

# Abstract

*This report contains documentation for the Deliverable 2.2.2:* Model Checking Tool for Artifact Interoperations (MOCAI) *of Work Package 2 after 24 months in the ACSI project. We present the requirements that serve as a guide for selecting the necessary features of the toolkit. We outline a methodology to model check declarative models of artifact-centric systems by translating GSM-based artifact-centric systems into a symbolic transition system used for symbolic model checking. A notable feature of our approach is that it is completely automatic. We implement the methodology in the GSMC model checker. The toolkit takes files directly from the web-based GSM engine Barcelona as input. We also provide preliminary results on the verification of artifact-centric systems using GSMC and demonstrate the applicability on an example from a real-world application.*

# Document History

| Version | Date | Comments |
|---|---|---|
| V0.1 | 01-04-2011 | Document created |
| V0.9 | 15-05-2011 | Internal review version (Iteration I) |
| V1.0 | 30-05-2011 | Final version (Iteration I) |
| V1.1 | 20-04-2012 | Initial draft (Iteration II) |
| V1.9 | 15-05-2012 | Internal review version (Iteration II) |
| V2.0 | 30-05-2012 | Final version (Iteration II) |

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

| **Acronym** | **Explanation** |
|---|---|
| ATL | Alternating-time Temporal Logic |
| CEGAR | Counterexample Guided Abstraction Refinement |
| CSP | Communicating Sequential Processes |
| CTL | Computation Tree Logic |
| DBM | Difference Bound Matrix |
| ECA | Event-Condition-Action |
| FD | First Draft |
| FRIS | Flanders Research Information Space Program |
| FSM | Finite-state Machine |
| GSM | Guard-Stage-Milestones |
| GUI | Graphical User Interface |
| ISPL | Interpreted Systems Programming Language |
| LFS | Local First Search |
| LTL | Linear Temporal Logic |
| MCK | Model Checking Knowledge |
| MCMAS | Model Checker for Multi-Agent Systems |
| MCTK | Model Checking Time and Knowledge |
| OBDD | Ordered Binary Decision Diagram |
| PAC | Prerequisite-Antecedent-Consequent |
| POEM | Partial Order Environment of Marseille |
| PROMELA | Process Meta Language |
| SMV | Symbolic Model Verifier |
| SPIN | Simple Promela Interpreter |

# 1 Introduction

One of the objectives of the ACSI project is the development of a verification toolkit that will implement the model checking techniques for artifact-centric systems defined in Task 2.1 (T2.1) of Work Package 2 (WP2). This document focuses on the specification for the toolkit.

System verification consists in proving that a design solution satisfies a set of requirement specifications. In particular, model checking [CGP99] is a verification technique where the system is represented by a mathematical model, and the requirement specifications are formulated in some (typically modal) logic. This reduces the verification task to checking whether the model satisfies particular logical formulae. A positive outcome guarantees that the system behaves as intended *before* it is deployed. Model checking has many advantages over other approaches, such as testing or theorem proving: it is fast, handles partial specifications, and in some cases produces counterexamples. It has been shown to be a suitable technique for the verification of reactive systems, distributed systems, and multi-agent systems [LQS08].

ACSI Interoperation Hubs are, potentially, huge IT environments supporting large number of services and involving many stakeholder organisations and independent users. It is therefore desirable to have a mechanism in place to ensure the validity of their designs, and model checking seems to be a promising technique to carry out this task.

Although model checking provides a powerful and efficient means of verification, it is generally limited to finite-state systems usually modelled as finite-state machines. However, many real systems, e.g., databases, do not belong to this class. In particular, artifact-centric systems rely heavily on their underlying databases and are specified by means of rich formalisms such as Guard-Stage-Milestones[1] (GSM) [HDF+11, HDM+11], a novel declarative formalism for describing lifecycles and processes. These two facts present a major obstacle for the verification of artifact-centric systems in that they give rise to infinite-state, non-trivial system behaviours. In order to overcome such obstacles, the model checking toolkit should fulfil the following key requirements. It should:

- be able to deal with suitable subclasses of infinite-state systems, possibly through approximations to finite models;
- implement a procedure for the transformation of declarative specifications of system lifecycles into state machines, i.e., the structures model checkers usually take as input;
- provide a formalism for requirement specification to be able to express relationships among data;
- accept inputs specified in suitable fragments of modal and/or first-order logics;
- ideally also provide a mechanism to model and verify time-related properties.

Building a new model checker is a very difficult task, and in addition, the points above pose a serious challenge themselves. We investigated whether it was necessary to start completely from the beginning since many model checkers already exist. And although none of them can be readily used for artifact-centric systems, we considered if some may serve as a suitable baseline for the development of our verification toolkit. However, the specific issues of GSM-based artifact-centric systems are so different from the technology currently used that it was preferable to implement a new model checker rather than extend an existing one.

[1]This formalism is being developed by ACSI in collaboration with IBM T.J. Watson laboratories and will be used for the implementation of ACSI Interoperation Hub in the following year.

The development of the toolkit has a close relationship with other results of the ACSI project. Above all with T2.1 (WP2), that will serve as a theoretical basis for the model checker. The toolkit will implement the verification techniques developed in T2.1, which in turn will be grounded on the ACSI Artifact Abstract Model ($A^3M$) defined in WP1. In particular, the toolkit will work with the GSM instantiation of $A^3M$. Lastly, both evaluation and testing of the toolkit will be carried out on the use cases identified in WP5. For this reason, the running example outlined in this document is the Flanders Research Information Space Program (FRIS) use case described in T5.1 (WP5).

The rest of the document is organized as follows: Section 2 presents the requirements placed on the toolkit for the verification of artifact-centric systems, along with a motivating example based on the use case from T5.1 (WP5). Section 3 reviews the state-of-the-art model checkers and discusses our decision to build a new model checker instead of using a baseline tool. Section 4 presents a methodology for encoding of GSM, generation of the transition relation, and the verification. Section 5 describes the input language, the system architecture, and the implementation details of the toolkit, which we call GSMC. Section 6 reports results on the verification of GSM-based artifact-centric systems using GSMC. Finally, section 7 concludes the document.

# 2 Requirements

In this section we describe the requirements placed on the tool for the verification and synthesis of artifact-centric systems. These requirements are derived from the work description for T2.2 (WP2) and are illustrated by a motivating use case from T5.1 (WP5). We begin by presenting the use case.

## 2.1 Use Case: FRIS

To illustrate the scenarios in which the ACSI framework may be applied, we briefly present the Flanders Research Information Space (FRIS) program of the Flemish government, which is one of the two pilot projects from WP5.

The FRIS program is centred around three strategic goals: (i) to accelerate the innovation chain by efficient and fast access to research information for all relevant stakeholders; (ii) to offer improved customer services; and (iii) to increase efficiency and effectiveness of the R&D policy. These goals are expected to be achieved through the change of the management process and service development. These services and processes should support stakeholders in the innovation ecosystem (see Figure 1) in facilitating their tasks. An important aspect of the FRIS program is making the involved data public.

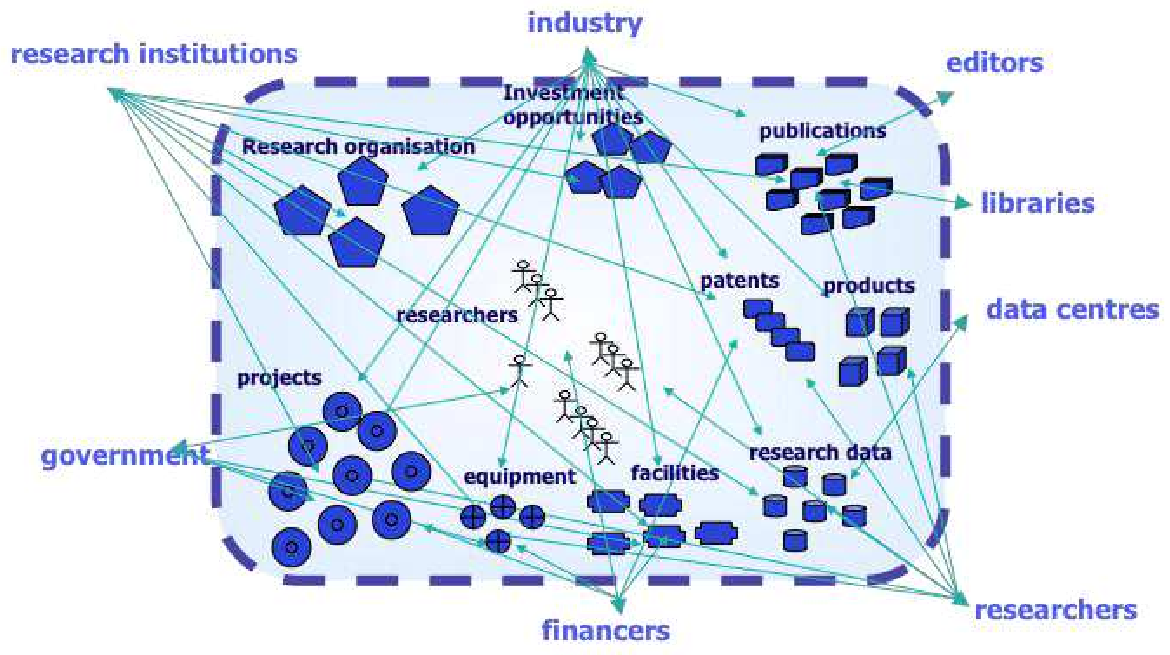

**Figure 1 – Overview of the FRIS ecosystem.**

The case study focuses primarily on the management of research programs. There are several agencies that provide funding to various research projects. Research programs are classified as either big, or small. Big projects are internationally oriented and usually count between fifty and sixty research proposals, while small projects are nationally oriented and can be well over one thousand research proposals. It is assumed that a funding agency has a permanent expert panel for the review process. This panel is the only entity involved in the reviewing of small research project proposals. For big projects, an external review board gets involved as well. The following stakeholders play a role in the FRIS use case:

**Funding Program Manager ($PM$).** A $PM$ is the manager of a research program. Her job starts with a Call for Proposals and ends when all the proposals are processed.

**External Reviewer ($ER$).** An $ER$ evaluates the research proposals that she is assigned to if she accepts an invitation for the external review from the PM.

**Principal Investigator ($PI$).** A $PI$ is the person responsible for a particular research proposal and the execution of the research project, if the proposal is successful.

**Project Officer ($PO$).** A $PO$ supervises research projects on behalf of the funding agency.

**Legal Representative ($LR$).** An $LR$ represents the $PI$ of a research proposal in a legal sense.

**Panel Member.** She is a member of the Expert Panel responsible for evaluating and monitoring the research proposals she is assigned to.

**Public.** The general public should be able to access publicly available information regarding current research projects.

The artifact-centric approach is well suited to this scenario as the stakeholders share a common environment within the ecosystem. A preliminary version of the FRIS use case consists of three key conceptual entities modelled as artifacts. This model allows for the creation and the management of a research program, the submission and evaluation of research proposals (submitted in response to a Call for Proposals related to the research program), and the execution and supervision of a research project resulting from a successful proposal. The three artifact types are:

**Program Evaluation Cycle.** This artifact type models research programs, including the evaluation of proposals but excluding the execution of projects. An instance is created by the $PM$ and ends when all proposals have been evaluated.

**Research Proposal Evaluation.** This artifact type deals with the data and lifecycle of research proposals. An instance of this type is created when a $PI$ registers in response to a Call for Proposals.

**Research Project Execution.** This artifact type models running research projects. The $PO$ supervises the execution of a research project, while the $PI$ periodically submits a report.

In the rest of this section we focus on the Research Proposal Evaluation (RPE) artifact type. The specification of the artifact is given in the GSM formalism [HDF$^+$11, HDM$^+$11].

The *Information Model* of RPE contains an integrated view of the relevant information about an RPE instance. It is essentially a set of attributes such as name of the proposal, $PI$'s contact information, requested budget, etc. The *Lifecycle Model* specifies the possible ways an RPE instance might progress through stages in response to events. Figure 2, where $\diamond$ represents guards, $\circ$ represents milestones, illustrates the stages of the RPE artifact type. The Negotiating stage is detailed in Table 1. Guards are quantified Boolean formulas that control when a stage becomes active. Milestones capture the objectives that can be achieved. The stakeholders are responsible for closing the stage by achieving some of these objectives.

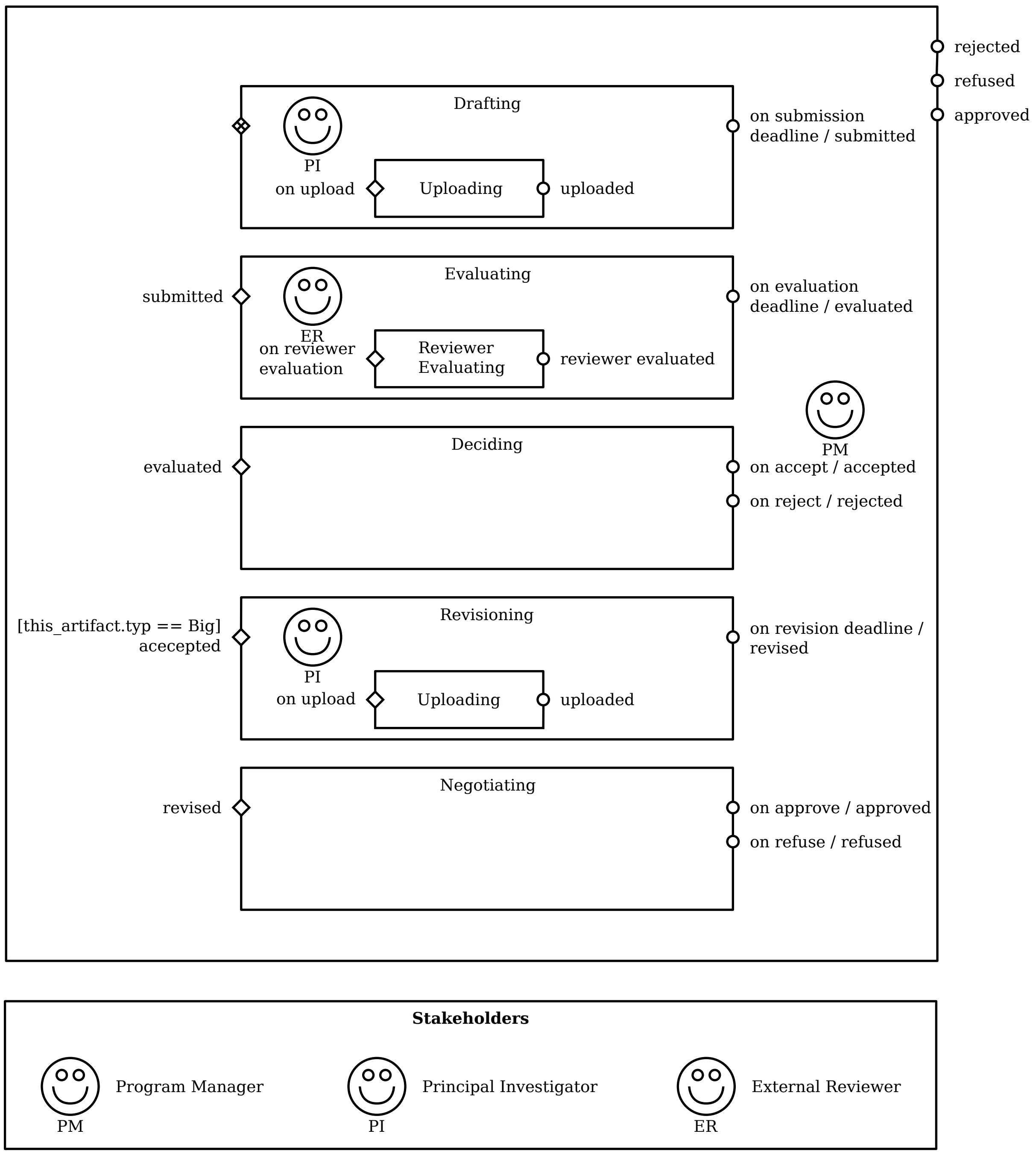

**Figure 2 – GSM model for Research Proposal Evaluation artifact type.**

| **Stage Name** | Negotiating |
|---|---|
| **Description** | Final negotiations regarding the proposal take place in this stage. |
| **Guards** | ProjectType=Big AND Deciding.Accepted=True AND Revisioning.Revised=True |
| **Milestones** | Negotiating.Approved \ onApprove<br>Negotiating.Refused \ onRefuse |
| **Stakeholders** | $PM$, $PO$, $PI$, and $LR$ |

**Table 1 – Overview of the Negotiating stage.**

There is a wide range of low-level and high-level properties that should be satisfied in a suitable design of the RPE artifact type. For instance, a simple property that is typically of interest is *reachability*: we may want to know if a research proposal can eventually be approved. This can be specified by a CTL formula[2] as follows:

$$EF\,approved.$$

More complex specifications can be expressed using combinations of CTL and epistemic operators. For example, we may want to verify that for a big proposal, whenever it is accepted and revised, everybody from the group $\Gamma$ of stakeholders consisting of $PM$, $PO$, $PI$, and $LR$ will eventually know that the final negotiation will be either approved or rejected. This property can be captured by the following modal formula:

$$AG((big \wedge accepted \wedge revised) \rightarrow AFE_\Gamma(approved \vee rejected)),$$

where $\Gamma = \{PM, PO, PI, LR\}$ is the group of stakeholders.

We may also be interested in the ability of the stakeholders to achieve certain temporal-epistemic goals. For example, we may want to check if $\Gamma$ can cooperate during the negotiation to eventually reach a consensus and approve the proposal. This can be expressed by the following ATL formula:

$$AG((big \wedge accepted \wedge revised) \rightarrow \langle\langle\Gamma\rangle\rangle F\,approved).$$

## 2.2 Requirements Listing

In this section we consider the requirements that a suitable model checking toolkit for the verification and synthesis of artifact-centric systems needs to fulfil. These requirements will serve as a guide for the development of the methodology and the actual implementation of the GSMC toolkit. Verifying full GSM has its difficulties and may not be viable to implement. Thus, we identify fragments that are essential and must be present in the final version of the model checker, whilst others provide advanced functionality if they are included but do not compromise the verification if left out.

**R1** *Release the toolkit as an open source software.*

[2]The modal operators used in this section have the following intuitive semantics. $EF\varphi$: there exists a path where $\varphi$ eventually holds; $AG\varphi$: along all paths $\varphi$ holds everywhere; $AF\varphi$: on all paths $\varphi$ eventually holds; $E_\Gamma\varphi$: everybody in group $\Gamma$ knows $\varphi$; $\langle\langle\Gamma\rangle\rangle F$: group $\Gamma$ can eventually enforce $\varphi$.

Firstly, many leading model checkers, such as NuSMV and MCMAS, are publicly available and distributed under various free software licenses. We expect that our toolkit will continue in this tradition.

The *raison d'être* of the model checker is to offer functionalities for the verification of artifact-centric systems. Thus, an important requirement is, of course, the capability of modelling such systems. WP1 introduces the *ACSI Artifact Abstract Model*, referred to as $A^3M$, which provides the conceptual and formal basis for the artifact layer of the Artifact Paradigm. Three instantiations of $A^3M$ were proposed: one based on finite-state machines (FSM) [CDH+08], one based on Proclets [vdABEW01], and one based on Guard-Stage-Milestones (GSM) [HDF+11, HDM+11]. We focus our efforts on GSM as it is planned to be the standard in the implementation of the ACSI Interoperation Hub. Since the implementation will be based on the prototype GSM engine, Barcelona, the model checker will be required to work with models specified by this engine. Barcelona evolved from the Siena system [CDH+08], and as such it captures GSM models directly in an XML format.

**R2** *Translating declarative specification of artifact lifecycles into finite-state machines.*

The major difference between GSM models for artifact-centric systems and the FSM models used by most static model checkers is that lifecycles specified in GSM are substantially more declarative than the FSM variants, although GSM is able to accommodate the FSM approach. The declarative style of artifact lifecycles is based primarily on conditions and rules, and supports hierarchy and parallelism within a single artifact instance. One way of translating a GSM model is by using the topological sort of the *polarized dependency graph* (see section 4) associated with the model, which specifies the well-formedness condition for the model.

**R3** *The support of first-order features in the modelling language.*

First-order logic and its fragments appear in several different contexts in artifact-centric systems. Crucially, a *sentry* for an artifact type is an antecedent having the form "on an event if a condition is satisfied", formally defined as expression **on** $\xi(x)$ **if** $\varphi(x)$, where *triggering event* $\xi(x)$ is an event expression and *condition* $\varphi(x)$ is a well-formed first-order formula over instances of this artifact type. Therefore, we need support for first-order constructs occurring in these conditions. In addition, according to WP1, the states of an artifact-centric system can be modelled as relational database instances. This implies that our verification framework needs to be able to express relationships among data, and this will in turn involve appropriate fragments of first-order logic. Temporal Logic formulas are also used in $A^3M$ for dynamic constraints that describe lifecycles of GSM artifact types.

**R4** *An implementation of abstraction techniques for verification of infinite-state systems.*

A major obstacle for model checking artifact-centric systems arises from the fact that no assumption can be made on the domains of values in the underlying database of the Information Model. Since these domains can be infinite, an artifact can have infinitely many states. Model checking infinite-state systems is one of the biggest challenges for the verification community. A complete verification algorithm for an arbitrary infinite-state system cannot be designed as the verification is, in general, undecidable.

One way of dealing with this problem is to place restrictions on the class of systems and/or on the specification languages and build a finite, abstract model of the original system under these restrictions. The abstract model can be then verified using standard model checking algorithms.

This method is called *abstraction*, and T2.1 (WP2) currently investigates abstraction techniques suitable for artifact-centric systems.

**R5** *The support of the specification languages identified by T2.1 (WP2).*

Another cornerstone requirement for the model checking toolkit concerns specification languages for artifact-centric systems. These are the languages used by the modeller to specify the properties of the system she wants to verify. Basic temporal logics, such as LTL and CTL, are commonly used as specification languages for simple properties such as: *reachability*, i.e., whether a given state can be reached from an initial state, *safety*, i.e., that something undesirable will never happen, and *liveness*, i.e., that something desirable will eventually happen. However, much richer and expressive specification languages, such as ATL, epistemic, and deontic logics, seem to be useful in the context of artifact-centric systems.

One of the objectives of T2.1 (WP2) is to explore the prospective specification languages that are grounded on models for artifact-centric systems developed in WP1. Once the suitable languages are identified, they will be amended for actual use by the model checker.

Note that requirements **R6** and **R7** are optional. They would provide advanced functionality but may not be fully implemented due to their complexity and the higher priority of the requirements above.

**R6** *A mechanism to model timeout behaviour and to verify timing properties.*

The notion of time is an important aspect of $A^3M$. On one hand, a timer may trigger events on timeouts, while, on the other hand, tasks performed by external actors take a certain amount of time. The actors typically carry out many tasks concurrently and we want to reason about the order in which they complete these tasks.

We consider two ways of modelling time: discrete time and dense time. In the first case, the clock values are natural numbers, whereas in the second case, they are real numbers. Model checking discrete time properties is easier [ACD93], although events that occur between two consecutive clock ticks are indistinguishable in time. In $A^3M$, discrete *logical timestamps* record the time when an event occurrence is incorporated into the system. Therefore, we can use the global clock to synchronise the system with actors and eliminate the necessity of dense time. However, introduction of the clock complicates symbolic model checking techniques, and so explicit representation of time may not be feasible to implement. Instead, certain specifications involving time can be implemented via specific propositions in ways to be explored.

**R7** *An implementation of game structures for synthesis.*

The last, optional requirement relates to *synthesis*. Whilst verification uncovers deviations from a given specification in an already built system, by synthesis one can build systems that are compliant by construction. However, synthesis of even a finite-state system is much more difficult in terms of computational complexity than its verification. This composition of systems can be modelled using *game structures* [Alu99]. A game structure is basically an extension of a Kripke structure, where state transitions result from choices (either turn-based, or concurrent) made by players.

# 3 State of the Art

In this section we provide a short review of several leading model checkers, discuss their suitability as a potential baseline for the verification toolkit, and explain our decision to build a new model checker rather than extend an existing one.

## 3.1 Model Checkers

Many model checkers for different purposes have been developed in recent years. In the following subsections we survey some of the tools that have the potential to be employed in model checking of artifact-centric systems.

### 3.1.1 NuSMV 2

Version 2 of NuSMV [CCG$^{+}$02] is a symbolic model checker originated from the reimplementation and extension of SMV, the first BDD-based model checker developed at CMU. The main novelty in the current version is the integration of model checking techniques based on propositional satisfiability (SAT) [BCCZ99].

The tool is written in ANSI C and provides a well structured, open and flexible platform for model checking. The different components and functionalities of NuSMV have been isolated and separated in modules, and interfaces between modules have been provided. This reduces the effort needed to extend the model checker. NuSMV has been designed as starting framework for the implementation and evaluation of new verification techniques. It has also been used as verification engine for tools in different application areas, ranging from formal validation of software requirements, to automated task planning.

NuSMV is able to process files written in an extension of the SMV language. In this language, it is possible to describe finite state machines by means of declaration and instantiation mechanisms for modules and processes, corresponding to synchronous and asynchronous composition, and to express a set of requirements in CTL and LTL. NuSMV can work in batch mode or interactively, with a textual interaction shell. It can also check real-time CTL specifications and computations, which specify discrete timing constraints.

NuSMV has been successfully adopted to model check both service-oriented architectures and multi-agent systems.

### 3.1.2 SPIN

SPIN [Hol03] is an open-source software tool originally developed at Bell Labs and used for the efficient formal verification of distributed software systems. The kernel has been implemented in ANSI standard C and the graphical interface has been developed using Tcl/Tk.

SPIN uses PROMELA as input language. PROMELA is a non-deterministic language, loosely based on Dijkstra's guarded command language with I/O operations based on Hoare's CSP language. SPIN also supports embedded C code as part of the model specification, which allows for direct verification of a C implementation of a system.

SPIN can be used for simulating behaviours of the system, as well as for exhaustive verification of the specified correctness, safety, and liveness properties. The theoretical foundation of its verification technique is based on the automata-theoretic approach. In addition to checking violation of assertions in PROMELA code, it can check if a property represented by an LTL formula is maintained by a system. The system, described in PROMELA, is modelled by finite-state automata. The negation of the LTL formula is modelled by a Büchi automaton. The synchronous product of the automata for the system and the property is generated, and if the language accepted by the product is empty, the property holds in the system. Otherwise, an error execution violating the property is reported. SPIN employs the explicit state model checking technique and generates the state space of a system on-the-fly. This means that it avoids the need to statically pre-compute the state space before verification.

The main limitation of SPIN is that it supports only LTL formulae. A richer, more expressive specification language is required for the verification of properties in artifact-centric systems. SPIN is also not a specialised model checker for multi-agent systems.

### 3.1.3 MCMAS

MCMAS [LQR09] is an OBDD based symbolic model checker specifically designed for multi-agent systems. It supports a number of modalities including CTL, ATL, and epistemic operators. The input language of MCMAS is Interpreted Systems Programming Language (ISPL), which has a rich specification capability to describe agents and multi-agent systems. Interpreted Systems [CGP99] provide the formal semantics of ISPL programs.

MCMAS is implemented in C/C++ and exploits the CUDD library [Som12] for BDD operations. The main components and the structure of the implementation can be summarised as follows:

- Input to the model checker is an ISPL file which is parsed and checked for syntax errors using the compiler tools Flex and GNU Bison. Various parameters are stored in temporary lists that are processed in the next step.
- The lists are traversed to build the OBDDs, created and manipulated using the CUDD library, for the verification algorithm. The set of reachable states is also computed.
- The formulae to be checked are parsed.
- Verification is performed by running the algorithm introduced in [RL04]. An OBDD representing the set of states in which a formula holds is computed.
- The OBDD for the set of reachable states is then compared to the OBDD corresponding to each formula. If they are equivalent, the formula holds in the model and the model checker produces a positive output. Otherwise, the model checker produces a negative output.
- MCMAS supports the creation of counterexamples for universal formulae (beginning with path quantifier $A$) to demonstrate how a universal formula is violated in a model, as well as witnesses for existential formulae (beginning with path quantifier $E$) if an existential formula holds in a model. MCMAS is also able to generate Graphviz files which represent counterexamples and witnesses for the provided model and for formulae.
- MCMAS also provides an Eclipse plugin which supports the creation of skeleton ISPL files, as well as syntax highlighting for them, and provides a graphical interface for performing the verification and examining the eventual counterexample/witness.

MCMAS is not the only tool specifically designed for model checking multi-agent systems. Other available model checkers are MCK [GvdM04], MCTK [SSL07], and Verics [NNP$^+$04]. However, MCMAS, to the best of our knowledge, is the only one implementing the ATL specification language.

### 3.1.4 UPPAAL

UPPAAL [BLL$^+$98] is an integrated tool environment for modelling, validation and verification of real-time systems developed in collaboration between the universities of Uppsala and Aalborg. The tool models real-time systems as networks of timed automata [Alu99] extended with data types such as bounded integers and arrays. It is therefore appropriate for systems where timing aspects are critical and that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. UPPAAL2k, the current version of UPPAAL, is a client/server application implemented in Java and C++ and available for Windows and Linux.

UPPAAL consists of three main modules: a description language, a simulator, and a model checker. The description language is a non-deterministic guarded command language with clock and data types, which serves as a modelling and design language for the system description. The simulator serves as a validation tool which provides graphical visualization and examination of possible dynamic behaviours of a system description in early design stages, thus allowing for fault detection prior to the more computationally demanding verification. The model checker exhaustively verifies dynamic behaviour of the system. It automatically checks safety and liveness properties by reachability analysis of the symbolic state space, in which states are represented by constraints, and also supports a simplified version of CTL, which does not allow nesting of path formulae.

The application of the on-the-fly searching technique and the DBM symbolic technique, which reduces verification problems to manipulation and solving of constraints, are crucial elements for the efficiency of the model checker. To facilitate debugging in case verification of a particular real-time system fails, the model checker may generate diagnostic traces that explain why a property is not satisfied by a system description. The diagnostic traces can be automatically loaded and graphically visualized using the simulator for the subsequent investigation.

### 3.1.5 POEM

POEM [NM10] is a modular model checking tool built to support several input languages, as well as several analysis and solving algorithms. Like many other model checkers, POEM uses an automaton-based internal model. It supports verification of both discrete systems and real-time systems. POEM has the basic structure of a compiler as it consists of three interconnected modules: a *frontend* performing syntactic and semantic analysis of an input model, a *core* performing a model transformation, and a *backend* that implements solving algorithms to verify the given property. The core then transforms the results into an XML file that is readable by a graphical interface. This kind of architecture is frequently used in model checkers and originally introduced in SPIN.

The frontend currently supports models written in UPPAAL or Verimag's IF2.0 language and transforms them into a common format. There are two backends. The original backend was based on a state space exploration with the underlying algorithms built around partial order reduction methods, and in particular Mazurkiewicz trace theory. For model checking of discrete systems, it employs an explicit model checking technique, LFS, with partial order reduction, and for real-time counterpart, it uses a DBM-like data structure accompanied by partial order reduction. The new SAT backend performs a bounded model checking: given a multi-threaded program and a reachability property, a SAT formula is constructed, such that it is satisfiable if and only if a state with the property can be reached by an execution of the program with up to some bounded number of multisteps.

The implementation language of POEM is Objective Caml. This choice is due to the advantages of functional programming languages for compiler writing, the efficiency of the language and the availability of non-functional features. However, the performance critical algorithms are delegated to external software written in C, and the GUI is implemented in Java. The disadvantage of POEM is that it does not check CTL or LTL formulae and only checks reachability.

### 3.1.6 Roméo

Roméo [GLMR05] is an integrated tool environment for modelling, validation and verification of real-time systems modelled as Time Petri Nets or Stopwatch Petri Nets, extended with parameters. It consists of computation modules written in C++ and a graphical user interface, written in Tcl/Tk, to edit and design Time Petri Nets.

As a design helper, Roméo implements on-the-fly simulation and reachability model-checking of Time Petri Nets and allows the early detection of some modelling issues during the conception stage. In addition to the on-the-fly simulation that makes simple design testing possible, Roméo provides an on-the-fly model-checker for reachability. A property over markings can be expressed and then the reachability of a marking satisfying the property can be tested. The tool returns a trace leading to such a marking if reachable. The model checker allows for verification of more complex properties expressed by observers that translate a property into a reachability test. This is the main method used to study the behaviour of a Time Petri Net.

In addition to on-the-fly model checking for reachability, Roméo implements different theoretical methods to translate Time Petri Nets into automata, timed automata or stopwatch automata. These models can be then verified against properties specified in temporal logics using an external model checker, such as UPPAAL. The first of the main methods for the translation is a structural transformation of a Time Petri Nets into a timed-bisimilar synchronized product of timed automata. The second method, a state space computation based translation, consists of the computation of the state class graphs that provide finite representations for the behaviour of bounded nets preserving their LTL properties.

The on-the-fly model checker of Roméo can only verify reachability in a Time Petri Net. For the richer specification language it translates the model into automata and uses UPPAAL as an external tool for model checking against these specifications. Thus Roméo suffers from the same shortcomings as UPPAAL.

## 3.2 Discussion

Building a completely new model checker is a complex and challenging task. The best tools currently available have been in development for many years, in some cases even decades. Table 2 summarises the features of the model checkers surveyed above. However, none of them can be applied to the verification of artifact-centric systems off-the-shelf due to the GSM declarative approach combined with the need of supporting data.

| Name | Modelling Language | Specification Language | Model Checking Technique | Timing Properties | Performance |
|---|---|---|---|---|---|
| NuSMV 2 | SMV | LTL, CTL, RTCTL | Symbolic (OBDD, SAT) | Discrete time | Very good |
| SPIN | PROMELA | LTL | Explicit | No | Medium |
| MCMAS | ISPL | ATL, CTLK | Symbolic (OBDD) | No | Good |
| UPPAAL | Timed automata, C | Subset of TCTL | Explicit & Symbolic (DBM) | Dense time | Good |
| POEM | Timed automata | Reachability | Explicit & Symbolic (LFS, DBM-like) | Dense time | Good |
| Roméo | Time Petri Nets | Subset of TCTL | Explicit | Dense time | Medium |

**Table 2 – Summary of model checkers.**

To alleviate the efforts of constructing an entirely new tool for GSM, D2.2.1 identified a plan of work that involved a compiler from GSM to a high-level modelling language that is supported by a model checker. The envisaged language was ISPL, the input language of MCMAS. Continued work on the topic during the second year showed, however, that there is a large disparity between the modelling language ISPL and GSM itself, which would lead to difficulties relating to requirements $R2$ and $R4$: Although data abstraction could be performed at the compiler level to produce a simplified ISPL model, the transformation of declarative GSM into FSM is expensive to model because the application of ground PAC rules must obey a specific order. Because MCMAS non-deterministically executes all enabled transitions in the ISPL model, this would require to add additional variables to resolve all non-determinism, which would lead to prohibitively large models. Alternatively, significant changes of the internals of MCMAS would be necessary. A source-to-source compiler from GSM to extended ISPL would also introduce problems in the verification process itself as counter-examples and witnesses generated by MCMAS correspond to the ISPL model and needed to be translated back into the original GSM specification to be understandable the users of the tool.

While the modelling language does not fit our requirements, the identified advantages of using MCMAS as model checking tool remain valid. First and foremost, complex, service-oriented distributed systems, such as artifact-centric systems, are very difficult to verify because of the large search space that arises from the many ways services can interact. The speed and memory cost when verifying a complex system is therefore crucial to the success of the verification. Symbolic model checkers like MCMAS are able to deal with a large state space because they represent states in compact structures rather than explicitly. Furthermore, we are interested in the verification of properties concerning not only the state of an artifact-centric system, but also the behaviour of stakeholders and their knowledge of the system. This means that the toolkit will need rich specification languages to be able to express these properties.

To capitalise on the experience from the MCMAS development at Imperial, a new tool, called GSMC, was developed, which directly supports GSM as input language but re-uses some verification algorithms from MCMAS. This allows us to concentrate the allocated manpower on GSM related features. This stand alone model checker still needs to meet the important criteria above, e.g., symbolic model checking techniques, support for multi-agent systems and a range of specification languages. Its main advantage is that it operates directly on GSM models produced by the Barcelona engine. The method of translating the declarative lifecycles into the transition relation of an FSM is described in Section 4. Given that we base GSMC loosely on MCMAS, some algorithms can be lifted from the existing model checker, notably the verification of CTL formulas once the transition relation is computed. However, the algorithms still need to be re-implemented and adjusted to the new checking tool and new approaches are required to handle the specifics of GSM. The main disadvantage of this approach is that it may not be possible to reproduce all the functionality of MCMAS in the time allocated to the ACSI project. The fundamental emphasis lies on the verification of artifact-centric systems and their interaction with the environment, while extended features, like counter-example computation, may not be implemented.

# 4 Methodology

In the following we present a novel methodology to verify the behaviour of artifact-centric systems in terms of its possible sequence of B-steps, which are defined as the smallest business relevant changes in the system. We refer the reader to [HDF$^+$11, HDM$^+$11] for an in-depth introduction to GSM.

## 4.1 Business Artifacts with GSM Lifecycles

We define a *GSM model* $\Gamma$ 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 the 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. This can be 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 for GSM has three equivalent formulations [DHV11]:

**Incremental** corresponds to the incremental application of the ECA-like rules and provides a natural, direct approach for implementation.

**Fixpoint** provides a top-down description of the effect of a single incoming event on an artifact snapshot.

**Closed-form** provides a characterization of snapshots and the effects of events using a first-order logic formula.

Although the GSM instantiation of $A^3M$ in WP1 uses the last one, we discuss only the incremental formulation here since our approach is based on this semantics, which is the most suitable for our model checking purposes. The incremental semantics of a GSM model $\Gamma$ 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 $\Sigma$, 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

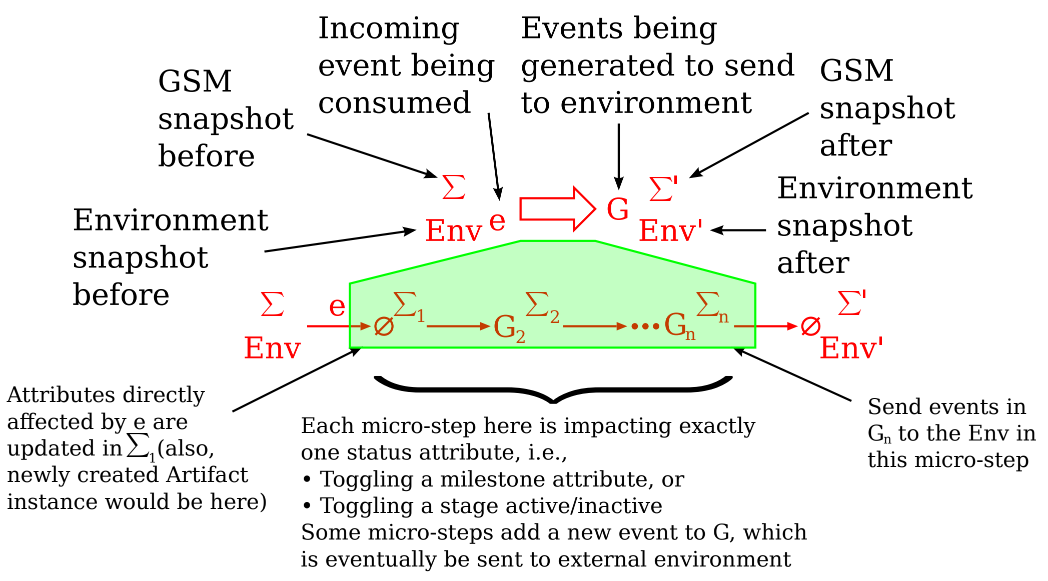
GSM snapshot before
Incoming event being consumed
Events being generated to send to environment
GSM snapshot after
Σ
Env
e
G
Σ'
Env'
Environment snapshot before
Environment snapshot after
Σ
Env
e
∅
Σ1
G2
Σ2
Gn
Σn
∅
Σ'
Env'
Attributes directly affected by e are updated in Σ1 (also, newly created Artifact instance would be here)
Each micro-step here is impacting exactly one status attribute, i.e.,
• Toggling a milestone attribute, or
• Toggling a stage active/inactive
Some micro-steps add a new event to G, which is eventually be sent to external environment
Send events in Gn to the Env in this micro-step

**Figure 3 – Incremental computation of GSM B-step.**

| 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** $-x.active_{S'}$ | $-x.active_S$ |

**Table 3 – PAC rule templates.**

Γ by *Prerequisite-Antecedent-Consequent* (PAC) rules that can lead to changes in guards and milestones. Figure 3 illustrates the incremental computation of a B-step.

The *abstract* PAC rules are listed in Table 3. 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 $\Sigma'$; the *consequent* ($C$) specifies the change to the value of a status attribute in the next snapshot $\Sigma'$ if the rule is relevant and if $A$ holds in $\Sigma'$. 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.

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 $\Sigma_i$ and $\Sigma_{i+1}$ is called a *micro-step*, whilst the B-step constitutes the transition from snapshot Σ to $\Sigma'$. The PAC rules are sequentially applied to $\Sigma_i$ until a fixed-point is reached. Each micro-step can generate an

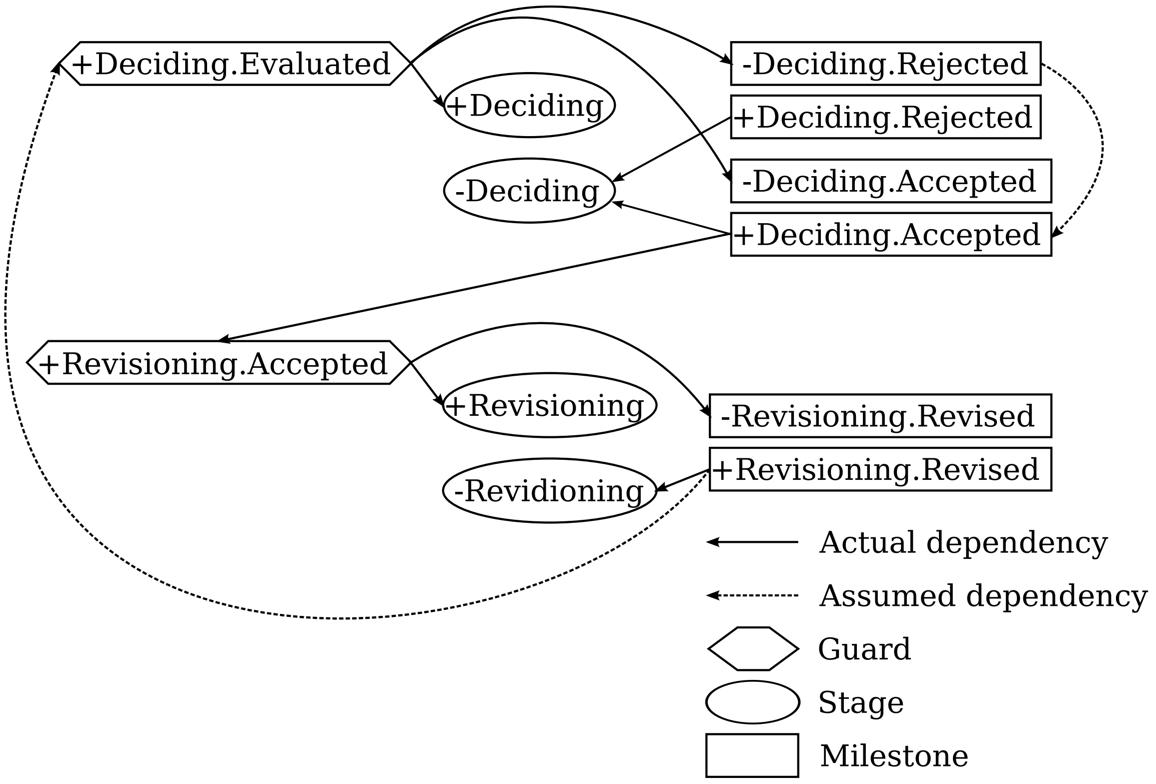

**Figure 4 – Partial polarized dependency graph of Research Proposal Evaluation.**

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 GSM model $\Gamma$. The set of nodes of the graph contains nodes for all guards, stages and milestones for each artifact type $R$ in $\Gamma$. The set of edges represents dependencies between individual nodes and it is based on ground PAC rules for $\Gamma$. If $DG(\Gamma)$ satisfies the acyclicity condition the model $\Gamma$ is *well-formed*.

Figure 4 shows part of this graph for Research Proposal Evaluation from the FRIS scenario. The arrows indicate dependency between guards, stages and milestones. + in front of a guard indicates that its sentry becomes true, $+/-$ in front of a stage indicates that the stage is open/closed, and $+/-$ in front of a milestone indicates that the milestone is achieved/invalidated. The dashed arrows are hypothetical dependencies, which would lead to a cycle.

As noted above, the incremental semantics of GSM is not directly amenable to symbolic model checking as it 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 divide the generation of a new snapshot into three phases illustrated in Figure 5:

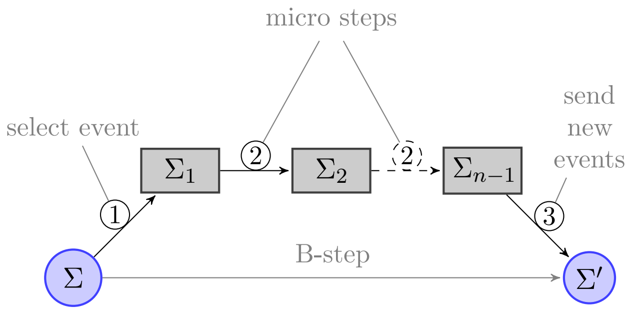

**Figure 5 – *B-steps* and *micro steps* of GSM.**

- *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. This 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 and how to use the transition releation to verify properies specified as CTL formulas. We start with giving a suitable encoding for a state, followed by the definition of a transition relation from the three phases given above.

## 4.2 Encoding

To perform verification, we need to capture the current state of an artifact-centric system with some additional information about its environment. A snapshot $\Sigma$ 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 $\Sigma$. For other pre-snapshots $\Sigma_i$ 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 $\Sigma$, $\overline{x}'\overline{e}'$ for the current pre-snapshot $\Sigma_i$, and $\overline{x}''\overline{e}''$ for the next pre-snapshot $\Sigma_{i+1}$. We use indices to access single elements in a vector.

## 4.3 Transition Relation

Using the encoding above, we define the transition relation as set of tuples $\delta \subseteq \Sigma \times \Sigma$ of start- and end-states for all possible B-steps. Alternatively, we also represent $\delta$ symbolically as Boolean function that evaluates to true for all $(\overline{x}\,\overline{e}, \overline{x}'\,\overline{e}') \in \delta$. This expression is easily built by computing the disjunction of all tuples in the set. The Boolean representation allows us to compute the successors for a set of states $\hat{\Sigma}$ in one step by 1) building the conjunction with the transition relation $\delta$ and 2) removing the current state variables and replace them by the next states: $\delta(\hat{\Sigma}) = (\exists_{\overline{x}\,\overline{e}} \hat{\Sigma} \wedge \delta)[\overline{x}\,\overline{e}/\overline{x}'\overline{e}']$. 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.

### 4.3.1 Execution of an Event

Phase ① generates the first pre-snapshot from a snapshot $\Sigma$ by picking a pending event $e_i$ and assigning the result of its execution to the attributes $x'$. The effect of a single event $e_i$ is given by $\delta_{e_i}$ below, where $e_i = \mathit{true}$ 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:

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

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:

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

The resulting transition relation produces all possible initial pre-snapshots $\Sigma_1$ for any set of snapshots $\Sigma$.

### 4.3.2 Execution of the PAC Rules

The main challenge in computing the updates to status attributes within a B-step is the interdependency 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 $\Sigma$. The antecedent is an expression over $\Sigma$ and the *next* snapshot $\Sigma'$. 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 $\Sigma$ and the pre-snapshot $\Sigma_i$. If A matches, the consequent updates the values for the next pre-snapshot while the values not touched by C remain as in $\Sigma_i$. The transitions generated by PAC rule $i$ with the unprimed variables for $\Sigma$, primed for $\Sigma_i$ and double primed for $\Sigma_{i+1}$ are given as:

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

The transition relation gives a new pre-snapshot $\Sigma_{i+1}$ for a given snapshot $\Sigma$ and pre-snapshot $\Sigma_i$. If we build the conjunction of $\delta_{r_1}$ with $\delta_{①}$, we get a formula that describes the first and second pre-snapshots following any snapshot $\Sigma$. 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 $\Sigma$ 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 $\Sigma$. 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.

### 4.3.3 Creating a new B-Step

The final phase ③ computes the resulting new snapshot $\Sigma'$ from $\Sigma$ and the last pre-snapshot $\Sigma_n$ by computing a new set of pending events and holding the data from $\Sigma_{n-1}$ 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 $\Sigma$ that are not in $\Sigma_n$:

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

An events is created when an atomic stage is activated during a B-step execution, i.e., the respective state attribute is set in $\Sigma_{n-1}$ but not in $\Sigma$. 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:

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

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 $\delta_{rem}$ and $\delta_{\mathcal{E}}$ is done analogously to $\delta_{r_i}$.

## 4.4 Agent Behaviour

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-centric system. To this end, we introduce a default agent that provides input to the artifact-centric 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-centric system reacts to these events. More formally, we give the transition relation of the agent $\delta_a$ as:

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

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

When verifying the model, we consider all sequences consisting of alternating steps of agent and artifact-centric system, starting from a single initial state $s_0$ with all data and events being zero. The joint transition relation $\hat{\delta}$ has the following formal definition:

$$\hat{\delta} = \delta_a \circ \delta = \exists_{\overline{x}'\overline{e}'} \delta_a \wedge \delta[\overline{x}''\overline{e}''/\overline{x}'\overline{e}'].$$

## 4.5 Verification

Symbolic model checking [BCM+90] 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. For clarity, we write $s$ for a state defined by variables $\overline{x}\,\overline{e}$. We use $\delta(s) = \{s' \mid (s, s') \in \delta\}$ to denote all *successors* of $s$. A run $\pi$ 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 [EH85] 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 $\varphi$ is given as

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

The semantics is defined inductively, where $\pi_s$ denotes all runs starting from a set of states

$s$. We say that a system $M$ with $s$ is a *model* of formula $\varphi$ (given as $(M, s) \models \varphi$) if:

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

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

Given a CTL formula, we compute the set of all the states in which the formula holds using the transition relations. If the set contains the initial state $s_0$ then the formula is true in the model and false otherwise. We now define the set of states $[\![\varphi]\!]$ in which formula $\varphi$ holds for the minimal set of CTL operators. For propositional atom $p$, negation $\neg\varphi$, and conjunction $\varphi_1 \wedge \varphi_2$, the sets $[\![p]\!]$, $[\![\neg\varphi]\!]$, and $[\![\varphi_1 \wedge \varphi_2]\!]$ have straightforward definitions:

$$
\begin{aligned}
[\![p]\!] &= \{s \mid p \in Ap(s)\}, \\
[\![\neg\varphi]\!] &= \{s \mid s \notin [\![\varphi]\!]\}, \\
[\![\varphi_1 \wedge \varphi_2]\!] &= [\![\varphi_1]\!] \cap [\![\varphi_2]\!].
\end{aligned}
$$

The more interesting cases arise when we deal with a formula involving temporal operators. For operator $EX$, we define the set $[\![EX\varphi]\!]$ as the set of all states which have a transition to a state in $[\![\varphi]\!]$. We call this set the pre-image of $[\![\varphi]\!]$:

$$[\![EX\varphi]\!] = \{s \mid \exists s' \in [\![\varphi]\!] \wedge s' \in \hat{\delta}(s)\}.$$

The set $[\![EG\varphi]\!]$ for operator $EG$ is computed as the *greatest* fixed-point of states that satisfy $\varphi$ and can proceed to a state that satisfies $\varphi$ as well. More specifically, starting with the set of all states, we take the intersection of the pre-image of this set and $[\![\varphi]\!]$. Pre-image $EX\ Y$ takes set $Y$ of states and returns the set of states which can make a transition into $Y$. We repeat the operation until we reach the greatest fixed point:

$$[\![EG\varphi]\!] = \nu Y.[\![\varphi]\!] \cap EX\ Y.$$

Similarly, the set $[\![E(\varphi U \psi)]\!]$ for operator $EU$ is computed as the *least* fixed point of states that satisfy $\varphi$ and can proceed to a state that satisfies either $\varphi$ or $\psi$. However, this time we start from the empty set, take the intersection of its pre-image, which is the empty set again, and $[\![\varphi]\!]$ and then we combine the result with $[\![\psi]\!]$ by taking the union of the two sets. We repeat the procedure until we reach the least fixed point:

$$[\![E(\varphi U \psi)]\!] = \mu Y.[\![\psi]\!] \cup ([\![\varphi]\!] \cap EX\ Y).$$

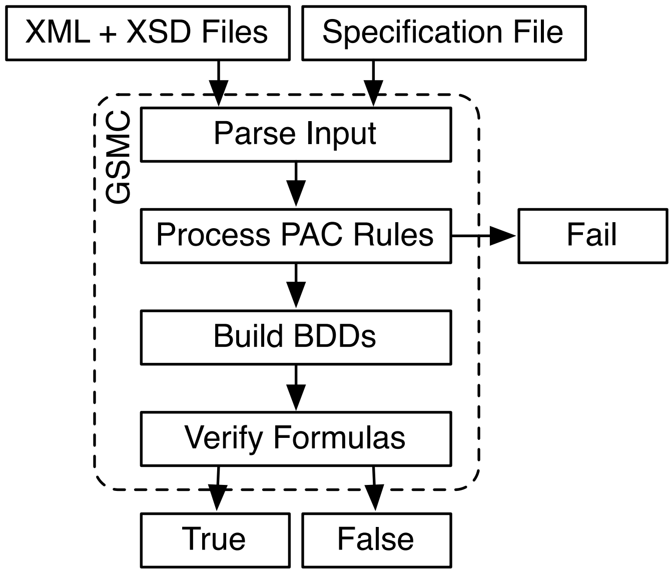

**Figure 6 – Architecture of GSMC.**

# 5 GSMC: System Specification

The methodology was implemented in the tool GSMC, a symbolic model checker that uses BDDs to represent the sets of states and transition relations. The tool is written in C++ with the CUDD library [Som12] for BDD operations. GSMC operates directly on models designed in the Barcelona editor. Such models of artifact-centric systems are stored in an XML based format and can be deployed on a Barcelona engine after 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 6 and consists of four phases:

**Parse Input:** The tool reads the inputs by using two parsers. An XML parser transforms the GSM model into an internal representation of the system, which consists of a hierarchy of objects representing the artifact types and stages in the system. Similarly, the specification parser creates parse trees of the formulas.

**Process PAC Rules:** Using the generated information about the available guards, stages, and milestones, the PAC rules templates from Table 3 are instantiated to generate the set of *grounded* PAC rules. The dependencies among the grounded PAC rules are analysed by constructing the dependency graph, which is used to perform a topological sort. If a dependency cycle is detected, the GSM model is not well formed; the verification is halted and a cycle that violates the acyclicity condition is produced and presented to the user.

**Build BDDs:** Starting from the sorted set of grounded PAC rules, BDD variables are allocated for the attributes and events of the model. This step also considers the dependencies among variables to reduce the size of the resulting BDDs. The tool then generates BDD representations of the PAC rules and constructs the transition relation of the artifact-centric system and the agent following the procedure described in Section 4.

**Verify Formulas:** The properties of the model from the specification file are verified one by one by traversing the parse tree of the formulas and computing the fixed points given in Subsection 4.5. All computations use the transition relation that was constructed in the previous phase and result in a BDD representation of the set of states in which a formula holds. If this set contains the initial state, the formula is true in the model; it does not hold otherwise. GSMC implements standard algorithms to compute the set for CTL formulas [BCM$^+$90].

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 the model are enumerated and can be compared for equality; 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. In the rest of the section we describe the implementation of the toolkit in more detail.

## 5.1 Barcelona GSM Models

In this section we briefly describe the input language for specifying GSM artifact-centric systems. We use the RapidXml parser for parsing Barcelona XML/XSD files containing GSM models. It is a small, fast, and free XML DOM-style parser written in C++ with good usability, portability, and reasonable W3C compatibility.

Before we list the most relevant elements and attributes used to describe a GSM model, note that the conditions of guards and milestones are specified using either Java Expression Language (JEXL) or Object Constraint Language (OCL). Both are declarative languages for describing rules that apply to GSM models. We currently implement JEXL only. JEXL has somewhat less expressive power than OCL. In particular, it does not support quantification over data.

The basic structure of a model is demonstrated using the excerpt of a model in XML in Figure 7. The structure of the model is a hierarchy, where the sections are delimited by the tags `<ca:Identifier ...>` and `</ca:Identifier>`. Elements without subsection are written as single tags `<ca:Identifier .../>`. The example contains the main sections of the model, but omits repetitions of sections of the same kind and some details. The root element of a Barcelona GSM application, called `ca:CompositeApplication` in Line 1, incorporates definitions of all artifact types and events. An artifact type is defined in element `ca:Component` in Line 3 with two attributes `id` and `name` containing the identifier and name of the artifact type. The child element `ca:InformationModel` (Line 4) specifies the artifact's information model. Importantly, the attribute `schemaUri` in its child element `ca:DataItem` includes the path to an external XML Schema Definition (XSD) file with the definition of the artifact type. All data attributes of the given artifact are declared as elements `xs:attribute` in this XSD file. Each date attribute has `name` and `type`, which can be either primitive (e.g., "xs:int", "xs:string", . . . ), or user defined. These non-primitive data types are defined using element `xs:complexType` and the definition is recursive in a sense that it is composed of attributes that can be complex types themselves.

The GSM lifecycle of an artifact is defined in Line 7 in element `ca:GuardedStageModel`. Children of this element represent top level stages of the artifact and are declared in elements `ca:Stage` with attributes `id`, `name`, and `description`. Stage guards are defined by `ca:StageGuard`. In addition to the usual attributes `id`, `name`, and `description`, stage guards have the attributes `eventIds` to lists events that are relevant for activating the stage, `expression` with the actual condition for the activation, and the optional attribute `language` that can be used to specify the language (OCL or JEXL) used in the condition.

```
<ca:CompositeApplication xmlns:ca=”http://siena.ibm.com/model/CompositeApplication”
    name=”FixedPriceProcurement”>
  <ca:Component id=”FPR” name=”FixedPriceRequest”>
    <ca:InformationModel id=”FPRInformationModel” rootDataItemId=”FPR”>
      <ca:DataItem id=”FPR” schemaUri=”FPR.xsd” rootElement=”FPR” />
    </ca:InformationModel>
    <ca:GuardedStageModel id=”FPR” name=”FixedPriceRequest” description=””>
      <ca:Stage id=”PreparingFPR” name=”PreparingFPR” description=””>
          <ca:StageGuard id=”GFPR1” name=”Guardstart1”
            expression=”!GSM.isStageCompleted(’PreparingFPR’)” eventIds=””/>
          <ca:Milestone id=”STS” name=”SubmittedToSuppliers” eventIds=””>
             <ca:Condition expression=”GSM.isMilestoneAchieved(’Launched’)”/>
          </ca:Milestone>
          <ca:Milestone id=”Abandoned” name=”Abandoned” eventIds=””>
             <ca:Condition expression=”GSM.milestoneAchievedOnEvent(’FPRAbandoned’)”/>
          </ca:Milestone>
          <ca:SubStage id=”Drafting” name=”Drafting” description=””>
             <ca:StageGuard id=”Guard17839” name=”Guard1”
                 expression=”!GSM.isStageCompleted(’Drafting’)” eventIds=””/>
              [...]
             <ca:Milestone id=”RRS” name=”RequesterReadySubmit” eventIds=”BlessSubmit”>
               <ca:Condition expression=”GSM.isEventOccurring(’BlessSubmit’)”/>
             </ca:Milestone>
              [...]
             <ca:SubStage id=”DraftingFPRData” name=”DraftingFPRData” description=””>
               <ca:StageGuard id=”Guard13171” name=”Guard1”
                  expression=”GSM.isEventOccurring(’InitiateFPR’)” eventIds=”InitiateFPR”/>
               <ca:Milestone id=”InitiateFPRDone” name=”InitiateFPRDone” eventIds=””>
                  <ca:Condition expression=”GSM.hasTaskCompleted(’DraftingFPRDataTask’)” />
               </ca:Milestone>
               <ca:Task id=”DraftingFPRDataTask” name=”DraftingFPRDataTask”>
                 <ca:Assign>
                   <ca:Mapping type=”set”>
                      <ca:Source sourceId=”InitiateFPRRequest” refType=”serviceRequest”
                          xPath=”InitiateFPRInputMessage/ProjectName”/>
                      <ca:Target targetId=”FPR” refType=”artifact” xPath=”FPR/ProjectName”/>
                       [...]
                   </ca:Mapping>
                 </ca:Assign>
               </ca:Task>
             </ca:SubStage>
              [...]
          </ca:SubStage>
           [...]
      </ca:Stage>
       [...]
    </ca:GuardedStageModel>
  </ca:Component>
     [...]
  <ca:EventModel id=”FPPEventModel” name=”FixedPriceProcurementEventModel”>
    <ca:Event id=”MCT” name=”ManuallyCloseTracking”>
      <ca:InputMsg id=”MCTReq” schemaUri=”MCTIn.xsd” rootElement=”MCTIn”/>
      <ca:OutMsg id=”MCTResp” schemaUri=”MCTOut.xsd” rootElement=”MCTOut”/>
    </ca:Event>
     [...]
  </ca:EventModel>
</ca:CompositeApplication>
```

**Figure 7 – Excerpt of a Barcelona model**

```
AG (   ParentFPR == 0
     || BiddingStyle != "FreeForm"
     || !GSM.isMilestoneAchieved('SupplierResponse', 'SelectedAsWinner')
     || EF GSM.isMilestoneAchieved('FixedPriceRequest', 'WinnerAssigned')
   )

EF (   ParentFPR == 1
     && GSM.isMilestoneAchieved('FixedPriceRequest', 'WinnerAssigned')
     && !GSM.isMilestoneAchieved('SupplierResponse', 'SelectedAsWinner')
   )

EF (   GSM.isMilestoneAchieved('FixedPriceRequest', 'RequesterReadySubmit')
     && EF GSM.isMilestoneAchieved('FixedPriceRequest', 'Drafted')
   )
```

**Figure 8 – Property formulas for GSMC**

Element `ca:Milestone` (e.g., in Lines 11 and 21) uses the attributes `id`, `name`, `description`, and `eventIds` as defined above and contains at least one element `ca:Condition` to describe an achieving condition of the milestone, and possibly a list of elements `ca:InvalidateCondition` to describe invalidating conditions of the milestone. Both elements have similar syntax as `ca:StageGuard` with attributes `id`, `name`, `description`, `expression`, and `language`.

A composite stage also contains a list of substages defined by elements `ca:SubStage` which have the same structure as element `ca:Stage`. An atomic stage may have a task specified by element `ca:Task` (Line 31). It has attributes `id` and `name` and can perform two different actions. Either a service, such as creation of a new artifact instance, can be invoked by an element `ca:Invoke` with an attribute `serviceDefinitionId`, or values can be assigned to some data attributes via element `ca:Assign`. Element `ca:Mapping` denotes the assignment to the data and its attribute `type` determines whether the current value will be overwritten or a new value will be added to a set. Child elements `ca:Source` and `ca:Target` then determine the actual value and the data attribute.

Finally, element `ca:EventModel` in Line 50 includes all incoming events that an agent in the environment can send to the artifact-centric system. Each such event has a corresponding element `ca:Event` with attributes `id`, `name`, and `description`. Its child element `ca:InputMessage` defines the payload of the event while `ca:OutputMessage` carries the system's response. Both elements have attributes `id`, `schemaUri` containing valid path to the XSD definition of the message, and `rootElement` defining the root element of the XSD file.

## 5.2 Input Language for Requirement Specification

The requirements for a model are supplied in a plain text file using a context-free language for logic formulas as shown in Figure 8. The file consists of a list of property formulas for temporal logic that are required to hold in the model. The parser for this language is implemented using the compiler generators Flex and GNU Bison.

We currently support `expressions` on data attributes and events, and operators for writing CTL formulas as presented in Subsection 4.5. In the following we give the grammar of the currently permitted operators in Backus-Naur form. We expect to extend the list with the

operators identified in the specification language developed in WP2 (T2.1).
Formally, expressions are defined as:

```
expression ::= constant
             | variable
             | expression aop expression
             | expression lop expression
             | GSM.isStageActive('artifactID', 'stageID')
             | GSM.isMilestoneAchieved('artifactID', 'milestoneID')
```

Where **aop** is an *arithmetic* operator, **lop** a logic operator, and **GSM.isStageActive** and **GSM.isMilestoneAchieved** are Boolean GSM operators to reason about the current state of an artifact with the identifier *artifactID*, one of its stages with identifier *stageID*, and a milestone *milestoneID* respectively. Type safety is checked by the parser, where currently supported types and operators are as follows:

**data types:** integer, double, string, Boolean constants and data attributes from XSD definition of the information model of an artifact;

**arithmetic operators (aop)** : *addition* (+), *subtraction* (−), *multiplication* (∗), *division*: (/), and *negation* (−);

**logic comparison operators (lop):** *equal to* (==), *not equal to* (! =), *less than* (<), *less than or equal to* (<=), *greater than* (>), and *greater than or equal to* (>=).

Using `expression` from above, a `formula` in the specification language has the following formal grammar:

```
formula ::= expression
          | ( formula )
          | formula && formula
          | formula || formula
          | ! formula
          | AG formula
          | EG formula
          | AX formula
          | EX formula
          | AF formula
          | EF formula
          | A ( formula UNTIL formula )
          | E ( formula UNTIL formula )
```

The grammar supports the usual logical connectives *and* (&&), *or* (||), and *not* (!). In addition, the CTL quantifiers over paths *all* (**A**) and *exists* (**E**), and the CTL operators *next* (**X**), *eventually* (**F**), *always* (**G**), and *until* (**U**) are supported.

## 5.3 Data

In GSM models, data types are only declared and used without formal definition or implementation of the corresponding operations on them. This has the advantage that they can include data attributes of arbitrary type and leave the actual implementation to a simulation engine like Barcelona. To perform a correct analysis in GSMC, however, these data types need to be mapped to implementations with a concrete semantics of all operations. GSMC currently implements three data types that can be used to represent Barcelona data: Boolean, enumeration, and bounded integer. In the following we give details on the implementation and the mapping between data in the models and GSMC. In line with requirement $R4$, we focus our efforts to further improve data handling for future versions of the toolkit.

A number of Barcelona data types have a direct correspondence in GSMC. A Barcelona `xs:boolean` data attribute with possible values *true* or *false*, for example, is directly mapped into a Boolean variable. The integer types `xs:int` and `xs:long` are mapped into bounded integer variables where the user can define the lower and the upper bounds for each variable. GSMC automatically adds an offset to the variables to optimise the internal representation as BDDs and corrects different offsets in computations. Any integer or long constants that appear in assignments to variables, however, must be in the specified range. The user can also define data attributes with predetermined lists of values. Such attributes are directly mapped into enumeration variables.

For data types without direct correspondence, some level of abstraction is required. We use C++ style type-casting to convert the `xs:double` data type to a bounded integer. This conversion between the two numerical types implies a loss of precision. For the `xs:string` data type, we enumerate all the string constants in the model and convert string data attributes to enumeration variables. Such variables can be assigned to any value that explicitly appears in the model. In addition, we define a special value $\perp$ that represents other possible unknown string constants from the environment. This basic abstraction allows us to assign values and compare strings for equality. More sophisticated string operations like concatenation are not supported yet and are reported by GSMC as error. Other data type are only rudimentary implemented. A variable of type `xs:date`, for example, is mapped into a Boolean flag that captures if it received a new value since initialisation of the artifact.

Encoding of the three data types requires the following number $n$ of BDD variables: $n = 1$ for a Boolean variable; $n = \lceil \log_2 |E| \rceil$ for an enumeration variable, where $E$ is the set of all enumeration values; and $n = \lceil \log_2(u-l+1) \rceil$ for a bounded integer variable, where $u$ is the upper bound and $l$ is the lower bound. GSMC supports equality checks for Boolean and enumeration variables. The full range of comparison operations is permitted for bounded integers. Furthermore, the bit operations ! (*not*), && (*and*), and $\|$ (*or*) are supported for Boolean variables and the arithmetic operations $+$, $-$, $*$, and $\backslash$ are supported for bounded integers. In case of arithmetic underflow or overflow, we raise a special Boolean flag to indicate this condition has occurred.

## 5.4 PAC Rules

To compute the transition relation of the model as outlined in Section 4, we instantiate the *abstract* PAC rule templates from Table 3 to model specific *ground* PAC rules. This section gives details of their encoding as BDDs, and how to compute the order in which they are applied during computation of the transition relation.

The BDD encoding of a ground PAC rule is computed from the encodings of its prerequisite, antecedent, and consequent respectively. The antecedents of the PAC-1, PAC-2, and PAC-3 rules are taken from the conditions of guards and milestones, while all others are determined by the abstract PAC rule templates. As an example, consider the following condition in some milestone achieving sentry:

*GSM.isEventOccurring('ReworkFPR') && GSM.isMilestoneAchieved ('Rejected').*

Assume that the *ReworkFPR* event at the current micro step is represented by the primed BDD variable $e'$, and the milestone *Rejected* is represented by the primed BDD variable $m'$. In this case, the BDD representation of the antecedent is simply $e' \wedge m'$. Other GSM operators need to access the the previous snapshot, which is represented by unprimed variables in our encoding. An example for such an operator is *GSM.isMilestoneAchievedOnEvent ('Rejected')*, which is only true if the milestone *Rejected* was achieved during the current B-Step and was false before. This is encoded as $\neg m \wedge m'$. The following GSM operators are currently implemented in GSMC and can be combined with the usual Boolean operators:

- *GSM.hasGroupOfAllRelatedArtifactsMilestoneBeenAchieved*
- *GSM.hasGroupOfAnyRelatedArtifactsMilestoneBeenAchieved*
- *GSM.hasRelatedArtifactMilestoneBeenAchieved*
- *GSM.hasTaskCompleted*
- *GSM.isRelatedArtifactStageActive*
- *GSM.isEventOccurring*
- *GSM.isMilestoneAchieved*
- *GSM.isStageActive*
- *GSM.isStageCompleted*
- *GSM.milestoneAchievedOnEvent*
- *GSM.RelatedArtifactMilestoneAchievedOnEvent*
- *GSM.stageActivatedOnEvent*
- *GSM.stageClosedOnEvent*

Once the prerequisite, the antecedent, and the consequent are encoded into BDDs $p$, $a$, and $c$ respectively, the BDD representation $r$ for the rule can be constructed in two steps. First, we consider cases when the rule is applicable and the consequent needs to be enforced. This is simply a conjunction of the three BDDs $r_1 := p \wedge a \wedge c$. Second, we account for cases when the rule is not applicable. In those cases the data must not be changed, which is expressed by $u$, which is the pairwise conjunction of all primed and double primed BDD variables that represent the artifact-centric system. The full expression for cases when the rule is not applicable is now given as $r_2 := \neg(p \wedge a) \wedge u$. Finally, we take the disjunction of the two BDDs $r_1$ and $r_2$ to obtain the BDD representation of the rule $r := r_1 \vee r_2$, which handles the full state space.

In the following we summarise the details for the different PAC rules from Table 3. We use $s$, $s'$ and $s''$ to encode the active flag of the corresponding stage for the previous snapshot, the current micro step and the next snapshot respectively. Similarly, $m$, $m'$ and $m''$ are used for the corresponding milestone.

**PAC-1** is instantiated for every guard in the model to activate the corresponding stage when the guard is fulfilled. The antecedent is taken from the condition of the guard. The prerequisite is encoded as $p := \neg s$ (stage was not open in the last snapshot) and the consequent as $c := s''$ (stage will be enabled in the next snapshot).

**Algorithm 1** `topologicalSort`($Q$)

```
while $|Q| \neq 0$ do
  $n \leftarrow Q$.pop()
  if $n.colour = 0$ then
    $n.colour \leftarrow 1$
    $P$.push($n$)
    while $|P| \neq 0$ do
      $n \leftarrow Q$.top()
      if $n$.nextChild() $\neq \emptyset$ then
        $n \leftarrow n$.nextChild()
        if $n.colour = 0$ then
          $n.colour \leftarrow 1$
          $P$.push($n$)
        else if $n.colour = 1$ then
          ERROR: circular dependency
        end if
      else
        $n.colour \leftarrow 2$
        $R$.push($n$)
        $P$.pop()
      end if
    end while
  end if
end while
return $R$
```

**PAC-2** is instantiated for every achieving sentry of every milestone in the model. We use the sentry as antecedent, encode the prerequisite as $p := s$ (stage was already open in the last snapshot), and the consequent as $c := m''$ (milestone achieved in the next snapshot).

**PAC-3** is instantiated for every invalidating sentry of every milestone in the model. Again, the sentry is used as antecedent. We encode the prerequisite as $p := m$ and the consequent as $c := \neg m''$.

**PAC-4** gives one ground rule for every guard-milestone pair of every stage in the model to invalidate the milestone if a stage opens. We encode the prerequisite as $p := m$ and the consequent as $c := \neg m''$. The antecedent has the same encoding as the antecedent of the corresponding PAC-1 rule that opens the stage.

**PAC-5** gives one ground rule for every milestone in the model to close the stage when the milestone is fulfilled. We encode the prerequisite as $p := s$, the antecedent as $a := \neg m \wedge m'$, and the consequent as $c := \neg s''$.

**PAC-6** gives one ground rule for every stage that is not a top-level stage in the model to close it when its parent $S_p$ is closed. Assume the parent is encoded using $s_p$, $s'_p$ and $s''_p$. We encode the prerequisite as $p := s$, the antecedent as $a := s_p \wedge \neg s'_p$ , and the consequent as $c := \neg s''$.

We build the dependency graph as described in section 4. The nodes correspond to guards, stages and milestone and the edges between them have associated ground PAC rules. Algorithm 1 describes the topological sort of this graph that determines the order in which the ground PAC rules are applied. The algorithm takes stack $Q$ of unsorted nodes and returns stack $R$ of nodes in their topological order, stack $P$ is auxiliary. Once the nodes are sorted, we examine the inbound edges of the nodes one by one and sort the ground PAC rules accordingly. After that, any ground PAC rule that has not been picked yet does not depend on any other rule and is pushed to the front.

## 5.5 Quick Reference Guide

GSMC is a command line application developed for Linux operating system. The toolkit can be downloaded from `http://vas.doc.ic.ac.uk/gsmc/gsmc.tar.gz`. To obtain the executable `gsmc`, extract GSMC source files with `tar` and type `make`. The command line options are displayed by running GSMC with `-h` or `--help` options. The following example executes verification of GSM model `fpr.xml` against specifications `formulas.txt`:

```
./gsmc -m fpr.xml -s formulas.txt
```

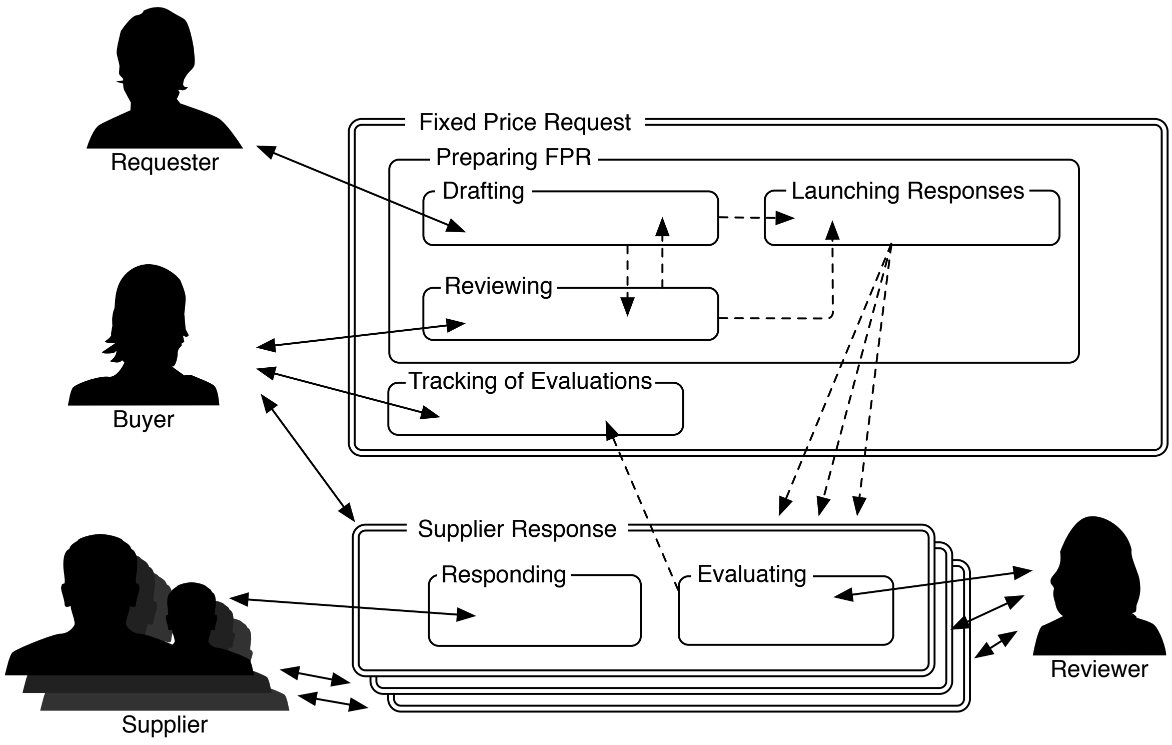

**Figure 9 – Overview of the Fixed Price scenario.**

# 6 Experimental Results

To evaluate our approach, we present preliminary results on the verification of artifact-centric systems using GSMC. Even though the evaluation is not yet done on the ACSI use cases from W5, we are able to verify properties of non-trivial GSM models produced by Barcelona engine. The reason why we have evaluated the tool neither on the case study nor on the Order-to-Cash example is that we did not have Barcelona models for these scenarios in the early stage of development. We stress that the verification is done automatically and directly on Barcelona models without the need for translating them into another modelling language.

## 6.1 Fixed Price Scenario

The Fixed Price contracting scenario [HDF$^+$11] 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 9 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-centric 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 10 and 11, 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.

## 6.2 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 10, says that whenever the 'Selected As Winner' milestone of the SR instance is achieved, then the 'Winner Assigned' milestone of the FPR instance can be eventually achieved. This is formally specified as follows:

$$\begin{aligned} AG\ (&MilestoneAchieved('SelectedAsWinner') \\ &\rightarrow EF\ MilestoneAchieved('WinnerAssigned'))) \end{aligned}$$

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-centric 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.

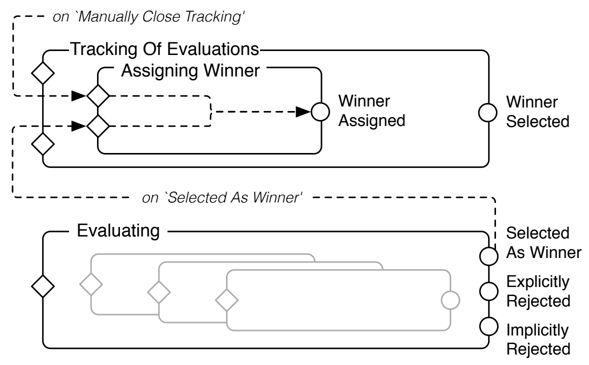

**Figure 10 – Structure of 'Evaluating' and 'Tracking Of Evaluations' stages.**

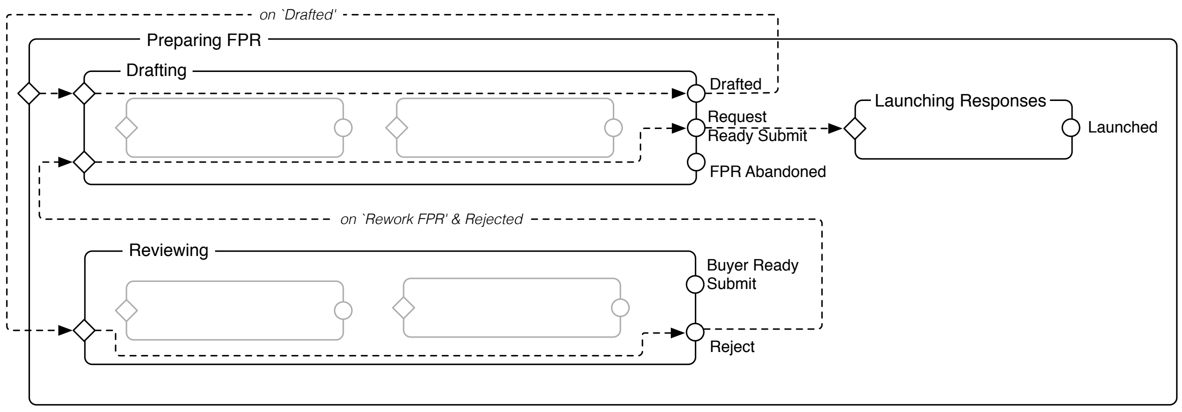

**Figure 11 – Structure of the 'Preparing FPR' stage.**

The third specification we verified is similar to the previous one and is also illustrated by Figure 10. 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:

$$\begin{aligned} AG\ (&MilestoneAchieved('WinnerAssigned') \\ &\rightarrow MilestoneAchieved('SelectedAsWinner')) \end{aligned}$$

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 11, 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:

$$\begin{aligned} AG\ (&MilestoneAchieved('RequesterReadySubmit') \\ &\rightarrow \neg EF\ StageActive('Drafting')) \end{aligned}$$

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.

## 6.3 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 $8 \times 10^{34}$ 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).

| Operation | Result | Memory | Time |
|---|---|---|---|
| Milestones can be Achieved | True | 100MB | 129.21s |
| Winner can be Assigned | True | 112MB | 26.52s |
| Winner Assignment Consistent | False | 84MB | 9.10s |
| Order Consistent | False | 73MB | 3.72s |

**Table 4 – Performance results.**

We verified the properties on a 64-bit Fedora 16 Linux machine with a 3.47GHz Intel® Core$^{\text{TM}}$ i5 processor and 8GB RAM. The GSMC constructed the transition relation in 3.76s using 72MB. Table 4 shows the results for the memory and CPU usage of the verification of the four properties discussed in this section. Note, that the first specification consists of 37 separate formulas. These results suggest that the verication time and memory requirements of the tool are small even for a realistic scenario with a large model.

# 7 Conclusions

In this document we reported progress in the development of the toolkit for the verification of artifact-centric systems after 24 months in the ACSI project. We presented the requirements that served as a guideline for selecting the necessary features of the toolkit. After the survey of several leading model checkers, we decided to capitalise on our experience from the development of MCMAS and build the new model checker, called GSMC.

We then 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 and we implemented the methodology in GSMC model checker. The toolkit takes files directly from the web-based GSM engine Barcelona as input.

We also provided preliminary results on the verification of GSM-based artifact-centric systems using GSMC. We demonstrated the applicability on Fixed Price contracting scenario, an example from a real-world application. The toolkit has shown to be capable of handling large models.

Although GSMC has already proven to be very helpful for validating GSM models, we investigate the implementation of abstraction techniques developed in T2.1 (WP2) to improve data handling in future versions of GSMC. A further important requirement is to check how the overall system behaves 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 supporting specification languages investigated again in T2.1 (WP2). We also work on the support of multiple instances per artifact type.

# Bibliography

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104/1, 1993.

[Alu99] R. Alur. Timed automata. In *Proceedings of CAV '99*, 1999.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of TACAS '99*, 1999.

[BCM$^+$90] 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.*, 98/2, 1990.

[BLL$^+$98] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of UPPAAL. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*, 1998.

[CCG$^+$02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of CAV '02*, 2002.

[CDH$^+$08] D. Cohn, P. Dhoolia, F. F. Heath, III, F. Pinel, and J. Vergo. Siena: From powerpoint to web app in 5 minutes. In *Proceedings of ICSOC '08*, 2008.

[CGP99] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking.* MIT Press, 1999.

[DHV11] 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.

[EH85] E.Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *JCSS*, 30/1, 1985.

[GLMR05] G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Roméo: A tool for analyzing time petri nets. In *Proceedings of CAV '05*, 2005.

[GvdM04] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of CAV '04*, 2004.

[HDF$^+$11] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, and P. Sukaviriya. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *Proceedings of WSFM '10*, 2011.

[HDM+11] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculin. Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events, 2011.

[Hol03] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[LQR09] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: a model checker for the verification of multi-agent systems. In *Proceedings of CAV '09*, 2009.

[LQS08] A. Lomuscio, H. Qu, and M. Solanki. Towards verifying contract regulated service composition. In *Proceedings of ICWS '08*, 2008.

[NM10] P. Niebert and J. Malinowski. SAT based bounded model checking with partial order semantics for timed automata. In *Proceedings of TACAS '10*, 2010.

[NNP+04] W. Nabialek, A. Niewiadomski, W. Penczek, A. Pólrola, and M. Szreter. Verics 2004: A model checker for real time and multi-agent systems. In *Proceedings of CS&P '04*, 2004.

[RL04] F. Raimondi and A. Lomuscio. Verification of multiagent systems via ordered binary decision diagrams: An algorithm and its implementation. In *Proceedings of AAMAS '04*, 2004.

[Som12] F. Somenzi. *CUDD: CU Decision Diagram Package - Release 2.5.0*, 2012.

[SSL07] K. Su, A. Sattar, and X. Luo. Model checking temporal logics of knowledge via OBDDs. *The Computer Journal*, 50(4), 2007.

[vdABEW01] W. M. P. van der Aalst, P. Barthelmess, C. Ellis, and J. Wainer. Proclets: A framework for lightweight interacting workow processes. *International Journal of Cooperative Information Systems*, 10(4), 2001.